Software Similarity Analysis

Similarity Analysis and Security

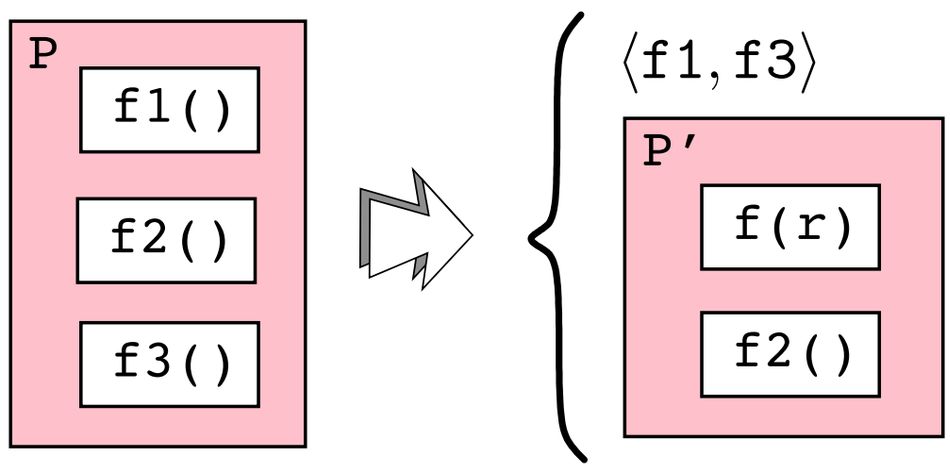
- Obfuscation, Tampering and Watermark protect SW through transformations
- Assume that the code has been released unprotected (performance, the adversary broke protection…)

Do programs A and B share common properties that would indicate that they also share a common origin?

- The answer to this question is important in:
  - ✓clone detection
  - ✓software forensics & code attribution
  - ✓plagiarism detection software
  - ✓birthmarking
  - ✓metamorphic malware detection

Clone Detection

- Duplicates may be the result of copy-paste-modify programming (better to abstract segment in a function call)
- Problem during *maintenance*: all copies need to be fixed
- Clone detection:
  - ✓locate similar pieces of code in a program
  - ✓clones are extracted out into functions



Clone Detection

- The programmer is not supposed to be malicious
- The code becomes "naturally" obfuscated because of the copy-paste-modify process
- The specialization process is usually simple: renaming of variables and replacement of literals with new values (more complex changes are unusual)
- Clone detectors are *SW maintenance* tools and they work either at the **source code level**, or they use an **high level representation** (syntax tree, dependence graphs).
- Clone detection could have a role in the SW protection scenario: We have seen that Skype was protected by adding hundreds of hash functions an that it was broken by manually discovering that these hash functions were very similar: a clone detector would have been able to discover these similarities?

Clone Detection

- DETECT(P, threshold, minsize)
  - ✓ Build a representation **rep** of P from which it is convenient to find clone pairs. Collect code pairs that are *sufficiently similar* and *sufficiently large* to warrant their own abstraction

$$
\begin{array}{l}
res \leftarrow \emptyset \\
rep \leftarrow \texttt{convenient representation of } P \\
\texttt{for every pair of code segments } f, g \in rep, f \neq g \texttt{ do} \\
\quad \texttt{if similarity}(f, g) \texttt{ > } threshold \texttt{ \&\&} \\
\qquad\qquad size(f) \geq minsize \texttt{ \&\& } size(g) \geq minsize \texttt{ then} \\
\qquad res \leftarrow res \cup \langle f, g \rangle
\end{array}
$$

Clone Detection

- DETECT(P, threshold, minsize)
  - ✓Break out the code-pairs found in the previous step into their own function and replace them with parametrized calls to this function

for every pair of code segments $f, g \in res$ do
  $h(r) \leftarrow$ a parameterized version of f and g
  $P \leftarrow P \cup h(r)$
  replace f with a call to $h(r_1)$ and g with $h(r_2)$

Software Forensics

Who wrote this piece of code?

- Security applications:
  - trace a piece of malware back to its *authors*
  - identify *illegal copies*
- It is important to extract the features of a program that are *unique* to a programmer's coding style
- It is usually performed at **source code** level since many programmer-specific features are stripped away by the compilations process (problem for malware detection, can relay on data structures, choice of compiler…)

Software Forensics

- DETECT(A, U, P, threshold)
  - Use a large corpus of programs from different authors to determine the set of characteristics C that have a high variability between authors and low variability between programs written by the same author
  - Collect sets of characteristics from the sample programs of each author and P

$sig \leftarrow \emptyset$
for each author $a \in A$ and code sample $s \in U[a]$ do
  $profile[a] \leftarrow$ characteristics $\mathcal{C}$ extracted from s
$sig \leftarrow$ characteristics $\mathcal{C}$ extracted from P

Software Forensics

- Determine which authors profile shares the most characteristics with P

for each author $a \in A$ do
    $sim \leftarrow$ similarity(sig, $profile[a]$)
    if $sim > thrashold$ then
        $res \leftarrow res \cup \langle a, sim \rangle$
return res sorted on similarity

Software Forensics

- Software Forensics studies also:
  - *Software characterisation:* draw conclusions about the psychological makeup and educational and cultural background of the author of a program based on aspects of its coding style
  - *Software discrimination:* tries to identify which pieces of a program were written by different authors

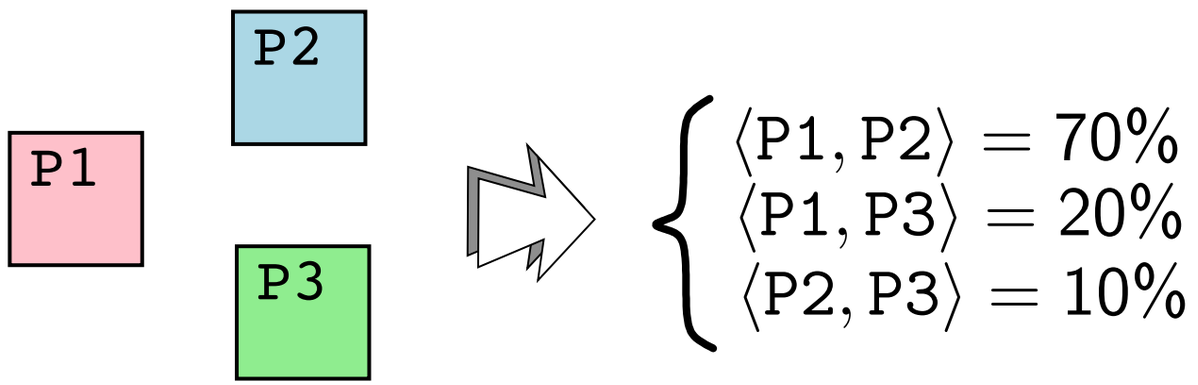
Plagiarism Detection

- *Plagiarized program*: is a program that has been produced by another program with trivial text edit operations and without detailed understanding of the program.

- hand in a verbatim copy of a friend's program
- make changes to the program to hide its origin
- "borrow" one or more difficult functions form a friend

Plagiarism Detection

- Computer science class: make a pair-wise comparison between all the programs handled in by the students



- The student needs the code to look reasonable
- General-purpose obfuscation - not OK
- Renaming `windowSize` to `sizeOfWindow` — OK
- Renaming `windowSize` to `x93` — not OK
- Replace a `while-loop` with a `for-loop` — OK
- Unroll a `loop` — not OK

Plagiarism Detection

$\textsc{Detect}(U, \mathit{threshold})$:

```
res ← ∅
for each pair of programs f, g do
    sim ← similarity(f, g)
    if sim > threshold then
        res ← res ∪ ⟨f, g, sim⟩
res ← res sorted on similarity
return res
```

Birthmark Detection

Detect similarities between programs in the following scenario:

Birthmarks are extracted from **executable code** (great need of analysis of malicious executables)

The adversary is more active:

- Copy one or more sections of the code P into Q
- Compile Q into binary or byte-code
- Apply semantics-preserving transformations, such as obfuscation and optimization, to Q before distributing it

Birthmark detection is the process of extracting properties of code that are invariant to common semantics-preserving transformations

Birthmark Detection

Unlike watermarking, birthmarking has *no embedding function* and *no secret key.*

* The birthmark is extracted from P statically

$$extract(P) \rightarrow b$$

or dynamically

$$extract(P, I) \rightarrow b$$

* We can decide if a birthmark is contained in another one

$$contains(b_P, b_Q) \rightarrow [0.0,\ 1.0]$$

* Detection is given by the combination

$$detect(Q,P) = contains(extract(Q), extract(P))$$

* The ability of the adversary in transforming P is modeled by

$$attack(P) \rightarrow P'$$

Successful attack: $detect(attack(P), P) < 0.5$

Similarity Measures

What does it mean for two peaces of code to be similar?

* Consider algorithms that extract “signals” from two or more programs and then compare these signals for similarity
* Signals could be:
  * sequences or set of extracted features
  * trees or graphs representing the structure of the program
* We need a way to compare them

Similarity Measures

* Similarity of **sequences of the same length**

Let f be a function that computes the future vectors of documents p and q of the same length: $f(p) = \langle p_1 \dots p_n \rangle$ and $f(q) = \langle q_1 \dots q_n \rangle$.

Hamming distance

$$distance(p, q) = |\{i | i \in [1, n], p_i \neq q_i\}|$$

Similarity

$$similarity(p, q) = 1 - \frac{distance(p, q)}{n}$$

Similarity Measures

* Similarity of **sequences of the different length**

The edit distance *distance*(p, q) between two sequences p and q is the minimum number of operations needed to transform p into q, using insertion, deletions, and substitutions. The similarity between p and q is defined as:

$$similarity(p, q) = 1 - \frac{distance(p, q)}{\max(|p|, |q|)}$$

Consider for example

$$\text{whiten} \xrightarrow{\textit{insert t}} \text{whitten} \xrightarrow{\textit{delete h}} \text{witten} \xrightarrow{\textit{replace k with w}} \text{kitten}$$

their similarity is

$$similarity(\text{whiten,kitten}) = 1 - \frac{3}{6} = \frac{1}{2}$$

Similarity Measures

* **set similarity**

Let $f(d)$ be a function that computes a set of features from a document d.
The similarity $similarity(p, q)$ between two documents p and q is defined as:

$$similarity(p, q) = \frac{|f(p) \cap f(q)|}{|f(p) \cup f(q)|}$$

The containment $containment(p, q)$ of p within q is defined as:

$$containment(p, q) = \frac{|f(p) \cap f(q)|}{|f(p)|}$$

Similarity Measures

* **set similarity**

Consider $f(p) = \{1, 3, 7, 8, 9, 11\}$ and $f(q) = \{2, 7, 9, 11\}$ then:

$$\begin{aligned} \textit{similarity}(p, q) &= \frac{|\{1, 3, 7, 8, 9, 11\} \cap \{2, 7, 9, 11\}|}{|\{1, 3, 7, 8, 9, 11\} \cup \{2, 7, 9, 11\}|} \\ &= \frac{|\{7, 9, 11\}|}{|\{1, 2, 3, 7, 8, 9, 11\}|} \\ &= \frac{3}{7} \end{aligned}$$

$$\begin{aligned} \textit{containment}(p, q) &= \frac{|\{1, 3, 7, 8, 9, 11\} \cap \{2, 7, 9, 11\}|}{|\{1, 3, 7, 8, 9, 11\}|} \\ &= \frac{|\{7, 9, 11\}|}{|\{1, 3, 7, 8, 9, 11\}|} \\ &= \frac{3}{6} \end{aligned}$$

Similarity Measures

* **graph similarity**

  * Adapt the edit distance metric to work on graphs (number of operations needed to transform a graph into another one)

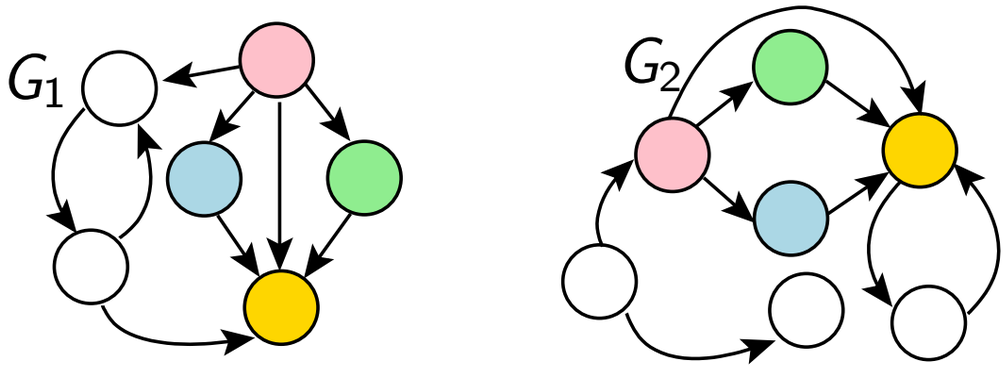
  * metrics based on *maximal common sub-graph*

Definition

Common subgraphs Let G, G_1, and G_2 be graphs. G is a *common subgraph* of G_1 and G_2 if there exists subgraph isomorphisms from G to G_1 and from G to G_2.
G is the *maximal common subgraph* of two graphs G_1 and G_2 ($G = mcs(G_1, G_2)$) if G is a common subgraph of G_1 and G_2 and there exists no other common subgraph G' of G_1 and G_2 that has more nodes than G.

Similarity Measures

- **graph similarity**
  - The colored nodes induce a maximal common sub-graph of G1 and G2 of four nodes



Similarity Measures

* **graph similarity**

Definition

Graph similarity and containment Let $|G|$ be the number of nodes in G. The *similarity*(G_1, G_2) of G_1 and G_2 is defined as

$$similarity(G_1, G_2) = \frac{|mcs(G_1, G_2)|}{\max(|G_1|, |G_2|)}$$

The *containment*(G_1, G_2) of G_1 within G_2 is defined as

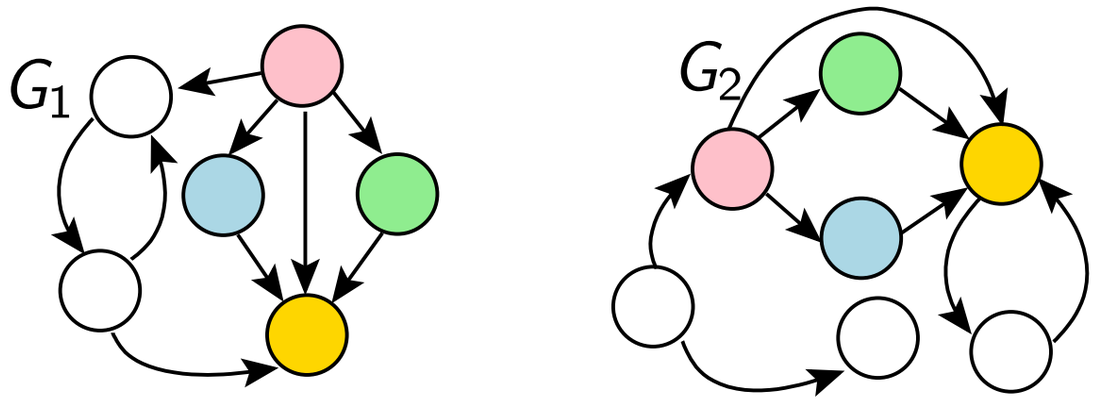
$$containment(G_1, G_2) = \frac{|mcs(G_1, G_2)|}{|G_1|}.$$

We say that G_1 is γ-isomorphic to G_2 if

$$containment(G_1, G_2) \geq \gamma, \gamma \in (0, 1].$$

Similarity Measures

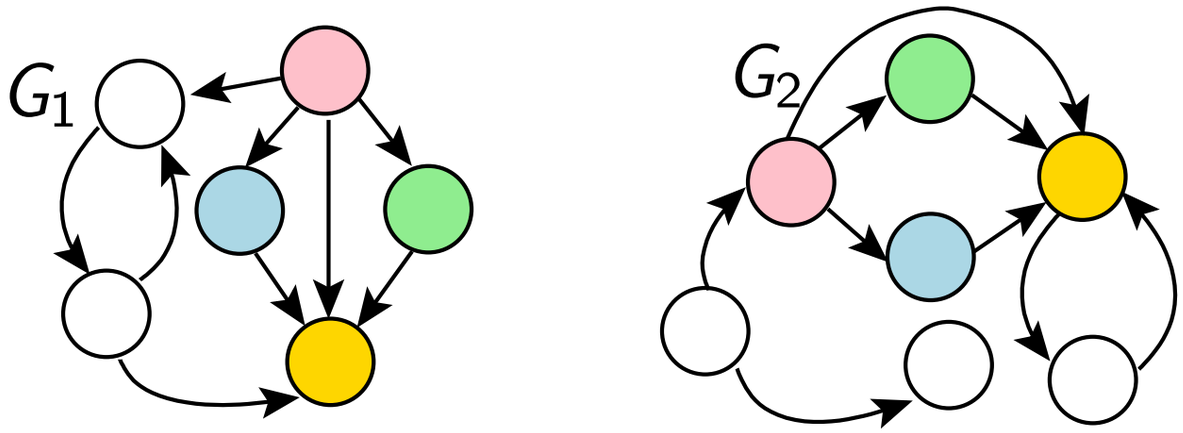
* **graph similarity**



- $similarity(G_1, G_2) = \frac{4}{7}$ and

Similarity Measures

* **graph similarity**



- $similarity(G_1, G_2) = \frac{4}{7}$ and
- $containment(G_1, G_2) = \frac{4}{6}$.

Similarity Algorithms

K-grams based similarity

k-grams hashes

- Comparing sets of k-grams is a common method to detect similarity:
  - plagiarism detection at source code level
  - authorship analysis at source code level
  - birthmark detection at executable code level

k-grams based similarity

- K-gram: continuous length k substring of the original document
- Consider a simple document A containing the string `yabbadabbadoo`

y	a	b	b	a	d	a	b	b	a	d	o	O
1	2	3	4	5	6	7	8	9	10	11	12	13

- by considering a window of size 3 over A we get the following set of 3-grams of A:
- `yab abb bba bad ada dab abb bba bad ado doo`

k-grams based similarity

- Let us consider a second document C containing the string `doobeedoobeedoo`

d	o	o	b	e	e	d	o	o	b	e	e	d	o	o
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- by considering a window of size 3 over C we get the following set of 3-grams of C:

- `doo sob obe bee eed edo doo oob obe bee eed edo doo`

- By comparing the 3-grams of A and C we observe that the only one in common is doo and that this appears one time in A and 3 times in C

k-grams based similarity

- The choice of *k* is very important
  - with *k* = 4 there is no similarity between A and C
- *k* should be such that common idioms of the document type have lengths less than *k*
- k-grams are insensitive to permutations
- For efficiency reasons
  - similarity algorithms usually store the hashes of *k*-grams
  - for large values of *k* they recur to rolling hash

k-grams based similarity

- Let us consider the following hash function for the example considered before:

hash(obe) = 15	hash(abb) = 2	hash(bee) = 16
hash(bba) = 3	hash(bad) = 4	hash(yab) = 8
hash(ydo) = 14	hash(eed) = 17	hash(byd) = 13
hash(doo) = 1	hash(ada) = 5	hash(edo) = 18
hash(ado) = 7	hash(coo) = 10	hash(dab) = 6
hash(oob) = 11	hash(oby) = 12	hash(sco) = 9

k-grams based similarity

- This leads to the following hashes for document A

yab	abb	bba	bad	ada	dab	abb	bba	bad	ado	doo
$A_0:8$	$A_1:2$	$A_2:3$	$A_3:4$	$A_4:5$	$A_5:6$	$A_6:2$	$A_7:3$	$A_8:4$	$A_9:7$	$A_{10}:1$

k-grams based similarity

- The number of hashes corresponds to the number of tokens of a document
- Impractical to store all of the hashes
  - Common approach: Keep only the hashes that are **0 mod p**, for some **p**, this may create a situation where there is a long gap between two stored hash values
- *Winnowing*: is the solution proposed in MOSS to avoid long gaps between hash values

MOSS: Source code plagiarism detection

- Sweep a window of size W over the sequences of hashes and keep the smaller hash in each window
- We cannot have a gap longer than W + k -1 between selected hashes
- Consider the windows of size 4 of the previous example

$(A_0:8$	$A_1:2$	$A_2:3$	$A_3:4)$
$(A_1:2$	$A_2:3$	$A_3:4$	$A_4:5)$
$(A_2:3$	$A_3:4$	$A_4:5$	$A_5:6)$
$(A_3:4$	$A_4:5$	$A_5:6$	$A_6:2)$
$(A_4:5$	$A_5:6$	$A_6:2$	$A_7:3)$
$(A_5:6$	$A_6:2$	$A_7:3$	$A_8:4)$
$(A_6:2$	$A_7:3$	$A_8:4$	$A_9:7)$
$(A_7:3$	$A_8:4$	$A_9:7$	$A_{10}:1)$

- The final set of hashes chosen from A becomes $\{A_1:2, A_2:3, A_6:2, A_{10}:1\}$

MOSS: Source code plagiarism detection

```
kGRAM(P,k,W)
```

- Let N be the length of P. Construct a list of *tokens* consisting of N - k + 1 substrings of P by sweeping a size k window over P
- Construct a length N - k +1 list of hashes by computing a hash over each string in *tokens*
- Construct a length |hashes| - W +1 list windows by sweeping a size W window over hashes
- Construct a set selected of hashes
  - selected = empty-set
  - for each window W in windows do
    - min = smallest rightmost hash in W
    - selected = selected U min
- Return selected

MOSS: Source code plagiarism detection

- *Software Plagiarism detection*: pairwise comparison of N programs, MOSS efficient also for large values of N since it postposes the quadratic step as long as possible
- Let us consider the following documents:
  - document `A` containing the string `yabbadabbadoo` with hashes $\{A_1:2, A_2:3, A_6:2, A_{10}:1\}$
  - document `B` containing the string `scoobydoobydoo` with hashes $\{B_0:9, B_1:10, B_2:11, B_6:1, B_7:11, B_{11}:1\}$
  - document `C` containing the string `doobeedoobeedoo` with hashes $\{C_0:1, C_1:11, C_2:15, C_6:1, C_7:11, C_8:15, C_{12}:1\}$

MOSS: Source code plagiarism detection

- Build a *index* over all hashes of all the documents.
- The index maps each hash value to the set of document locations where they occur

1	$A_{10}:1$	$B_6:1$	$B_{11}:1$	$C_0:1$	$C_6:1$	$C_{12}:1$
2	$A_1:2$	$A_6:2$				
3	$A_2:3$					
9	$B_0:9$					
10	$B_1:10$					
11	$B_2:11$	$B_7:11$	$C_1:11$	$C_7:11$		
15	$C_2:15$	$C_8:15$				

MOSS: Source code plagiarism detection

- Hash the document again using the index to construct, for every document, a list off matching hashes in the other documents

A	B_6:1	B_{11}:1	C_0:1	C_6:1	C_{12}:1	
B	A_{10}:1	C_0:1	C_6:1	C_{12}:1	C_1:11	C_7:11
C	A_{10}:1	B_6:1	B_{11}:1	B_2:11	B_7:11	

MOSS: Source code plagiarism detection

- Note that until now we have avoided any quadratic behavior!
- The final step is for each pair of documents, to traverse their list of hashes and extract those they have in common

[A,B]	B_6:1	B_{11}:1	A_{10}:1						
[A,C]	A_{10}:1	C_0:1	C_6:1	C_{12}:1					
[B,C]	C_0:1	C_6:1	C_{12}:1	C_1:11	C_7:11	B_6:1	B_{11}:1	B_2:11	B_7:11

- The document pairs with the longest list of hashes are most likely to be instances of plagiarism

Source Code Preprocessing

- To thwart detection software plagiarist might use simple transformations such as
  - Reordering functions
  - Renaming variables
  - Reformatting the code
- k-grams analysis takes care of coarse-grained reordering but some *preprocessing* that *canonicalize the code* is necessary to handle variable renaming and reformatting

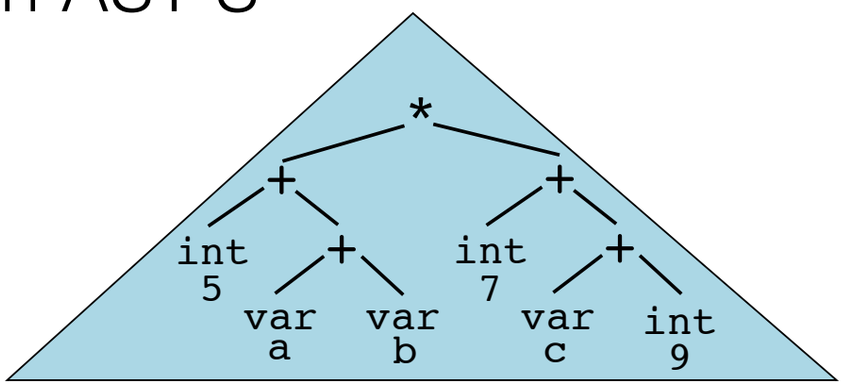
AST-based similarity

AST-based clone detection

Look for clones in this program

```
(5 + (a + b)) * (7 + (c + 9))
```

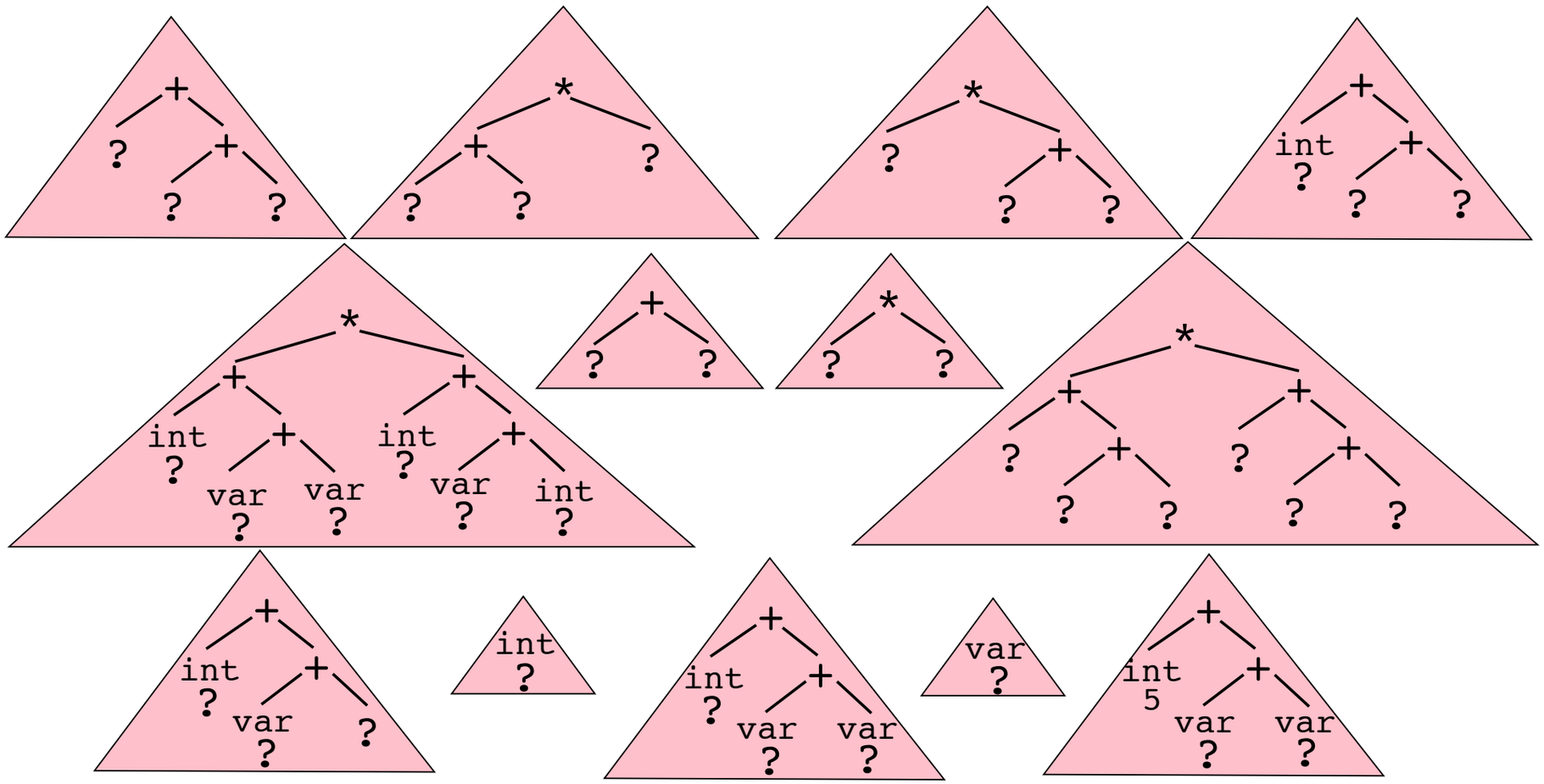
Parse and build an AST S



AST-based clone detection

- Construct all *tree patterns*
- A tree pattern is a subtree of S where one or more subtrees have been replaced with a *wildcard*
- We'll color the ASTs themselves blue and the tree patterns pink

AST-based clone detection

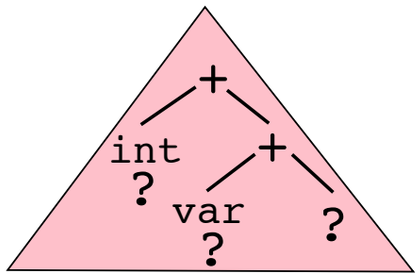


AST-based clone detection

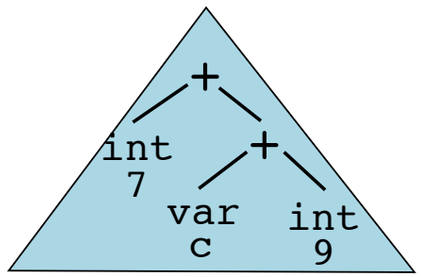
- What is a clone in the context of an AST?
- Simply a tree pattern for which there is more than one match!
- Which patterns would make a good clone?
  - has a large number f nodes
  - occurs a large number of times in the AST
  - has a few holes

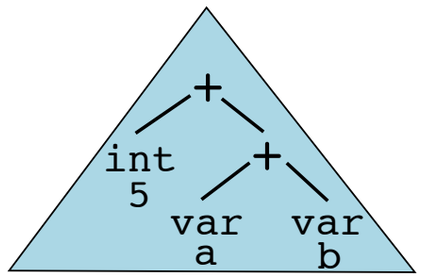
AST-based clone detection

This pattern seems like it might make a good choice



It matches two large subtrees of S





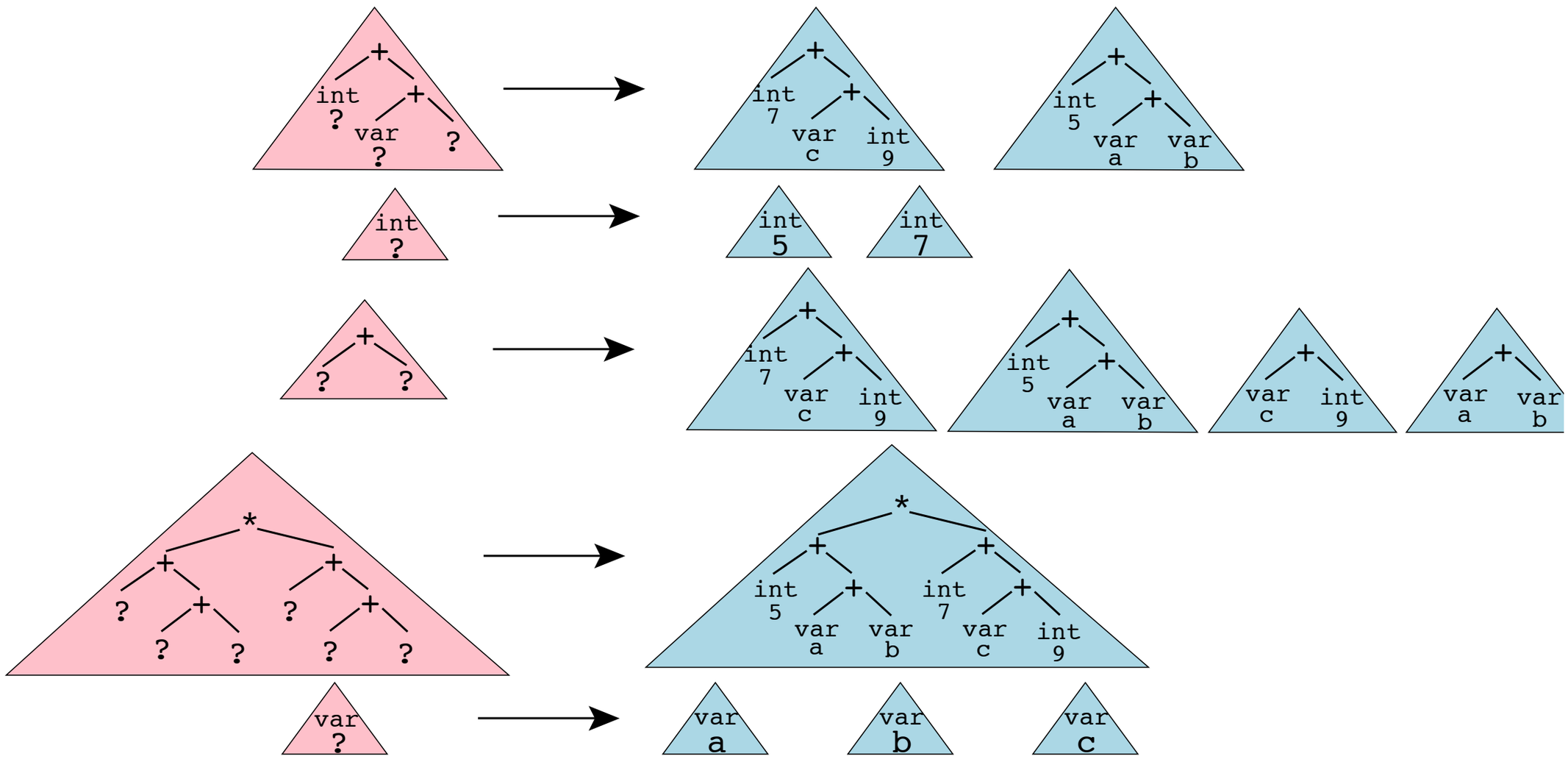
AST-based clone detection

- Now you can extract the clones and turn them into macros

```
#define CLONE(x,y,z) ((x)+((y)+(z)))
CLONE(5,a,b) * CLONE(7,c,9)
```

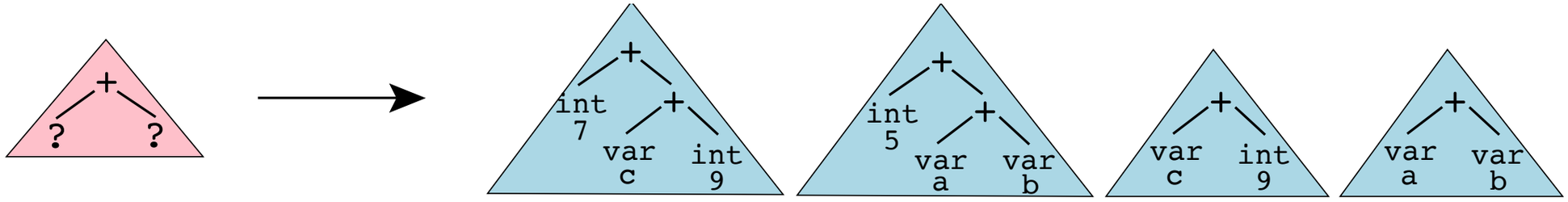
- Build a clone table, a mapping from each pattern to the locations in S where it occurs
- Sort the table with large patterns, most number of occurrences, fewest number of holes first!

AST-based clone detection



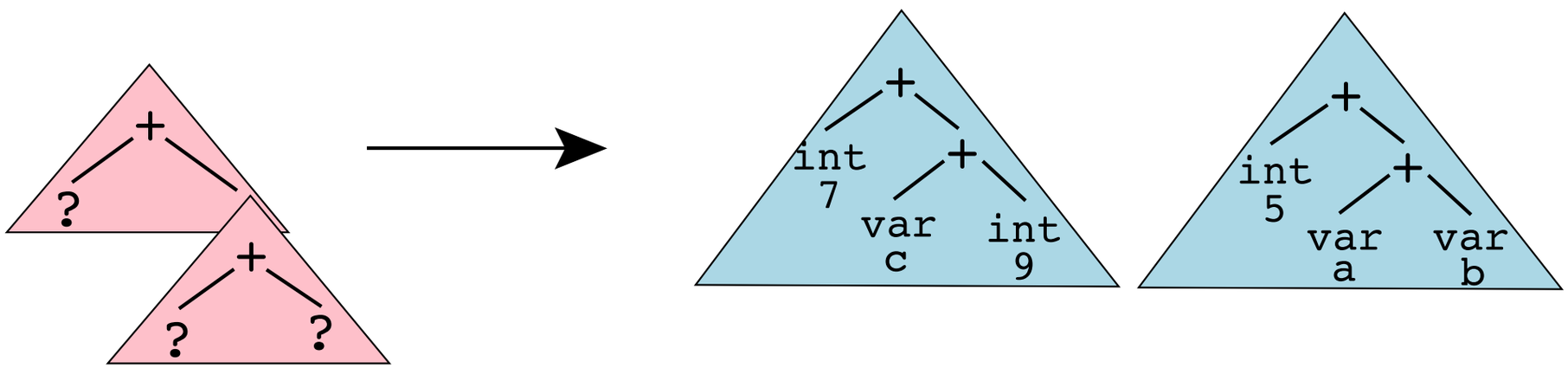
AST-based clone detection

- Won't scale: exponential number of tree patterns
- Idea: iteratively grow larger tree patterns from smaller ones
- Step 1:



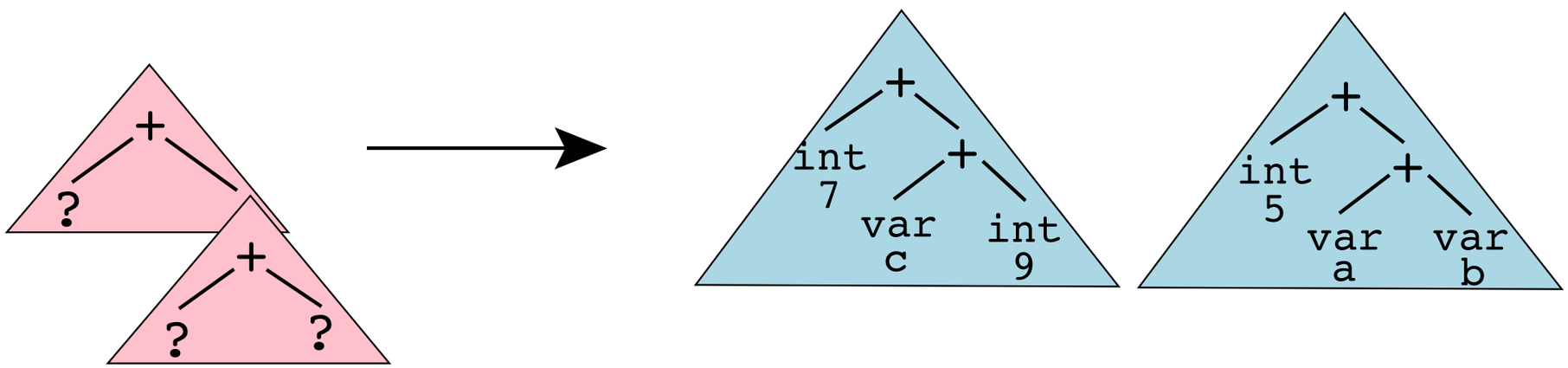
AST-based clone detection

- We specialize, and the new pattern becomes larger (but only has 2 matches)

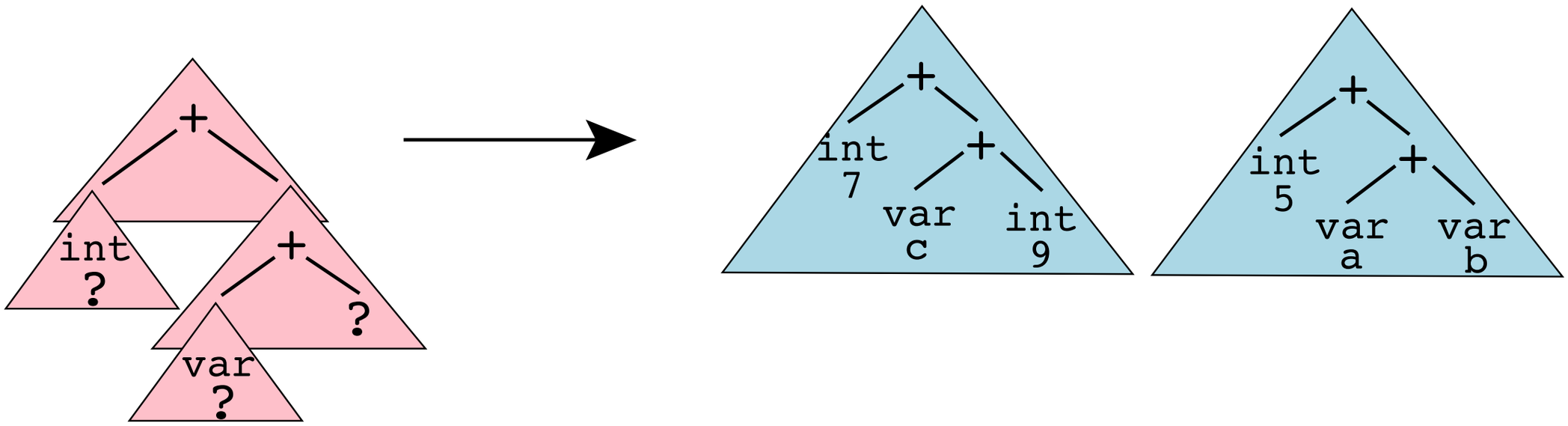


AST-based clone detection

- We specialize, and the new pattern becomes larger (but only has 2 matches)



- After two more steps of specialization we are done



Graph-based similarity

Graph-based analysis

Programs are graphs:

- Control flow graphs
- Dependence graphs
- Inheritance graphs
- Can program similarity be computed over graph representations of programs?

Graph-based analysis

Unfortunately:

- Sub-graph isomorphism is NP-complete
- Fortunately, graphs computed form programs are not general graphs
- Control flow graphs will not be arbitrarily large
- Call-graphs tend to be very sparse
- Heuristics can be very effective in approximating subgraph isomorphism

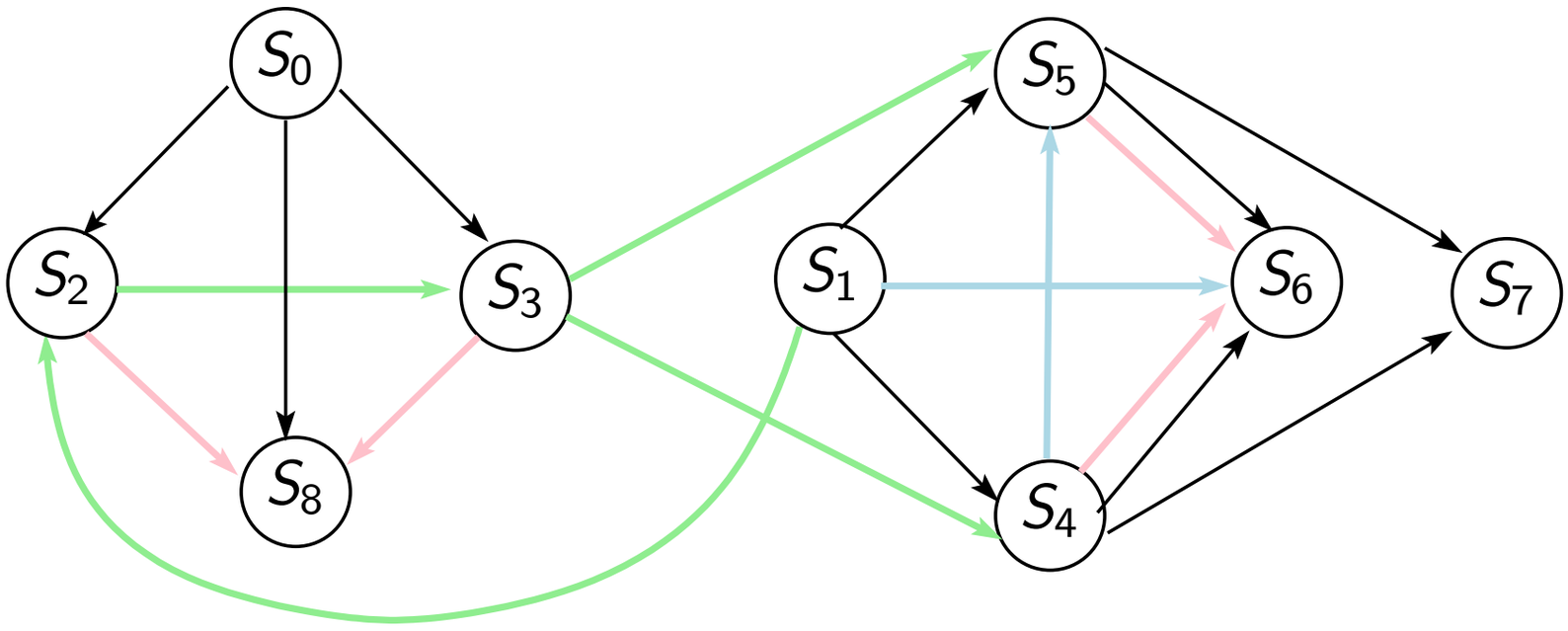
PDG-based clone detection

- The nodes of a PDG are the statements of a function
- There is an edge m ⟶ n if
  - n is data-dependent on m, or
  - n is control-dependent on m
- Semantics-preserving reordering of the statements of a function won't affect the graph

PDG-based clone detection

```
S0 : int k = 0;
S1 : int s = 1;
S2 : while (k < w) {
S3 :     if (x[k] == 1)
S4 :         R = (s*y) % n;
       else
S5 :         R = s;
S6 :     s = R*R % n;
S7 :     L = R;
S8 :     k=k+1;
   }
```

PDG-based clone detection



PDG-based clone detection

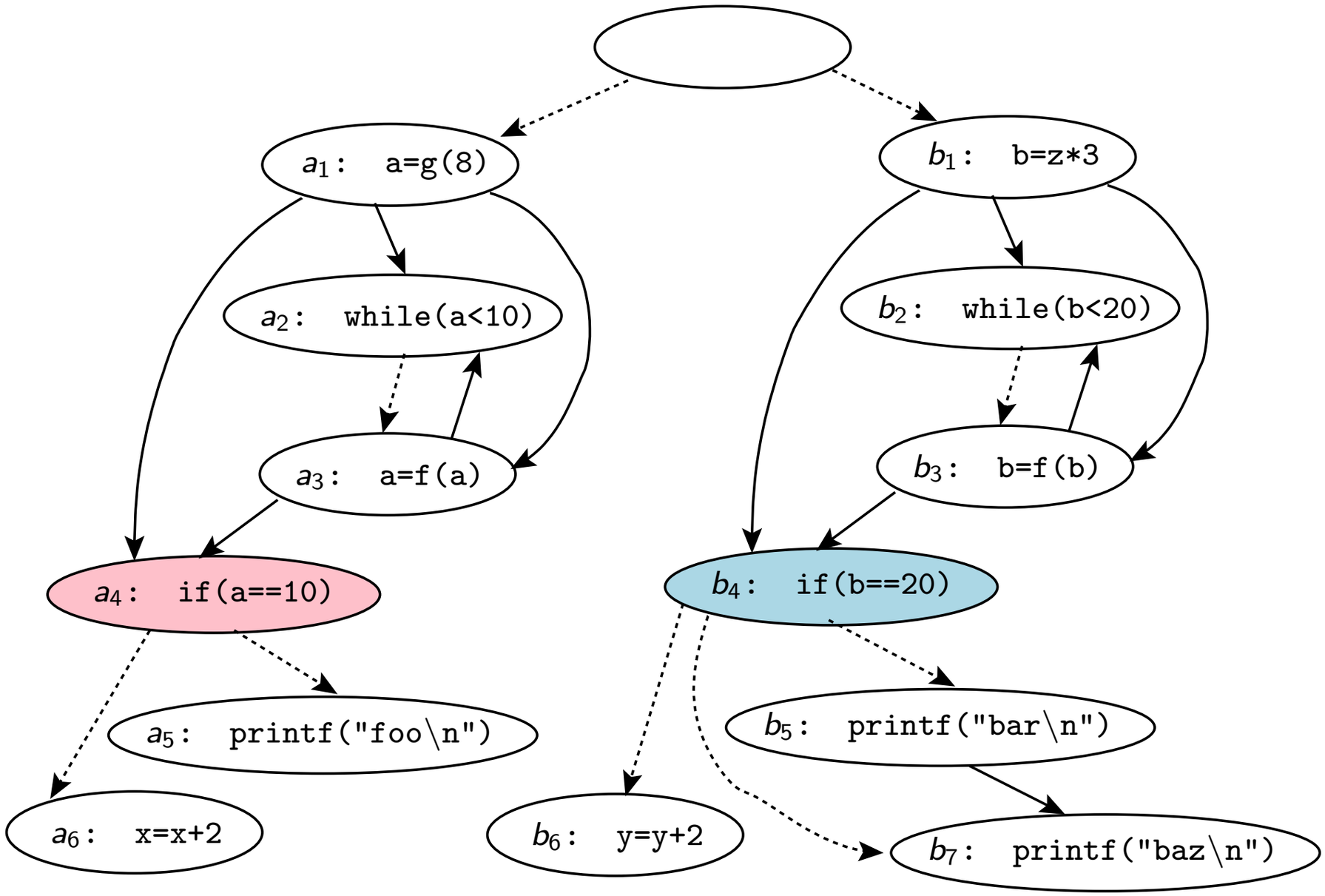
- Build a PDG for each function of the program
- Compute two isomorphic subgraphs by slicing backwards along dependency edges starting with every pair of *matching nodes*
- Two nodes are matching if they have the same syntactic structure
- Repeat until no more nodes can be added to the slice

PDG-based clone detection

```
a1:  a = g(8);
b1:  b = z*3;
a2:  while(a<10)
    a3:  a = f(a);
b2:  while(b<20)
    b3:  b = f(b);
a4:  if (a==10) {
    a5:  printf("foo\n");
    a6:  x=x+2;
}
b4:  if (b==20) {
    b5:  printf("bar\n");
    b6:  y=y+2;
    b7:  printf("baz\n");
}
```

- Two similar pieces of code have been intertwined within the same function

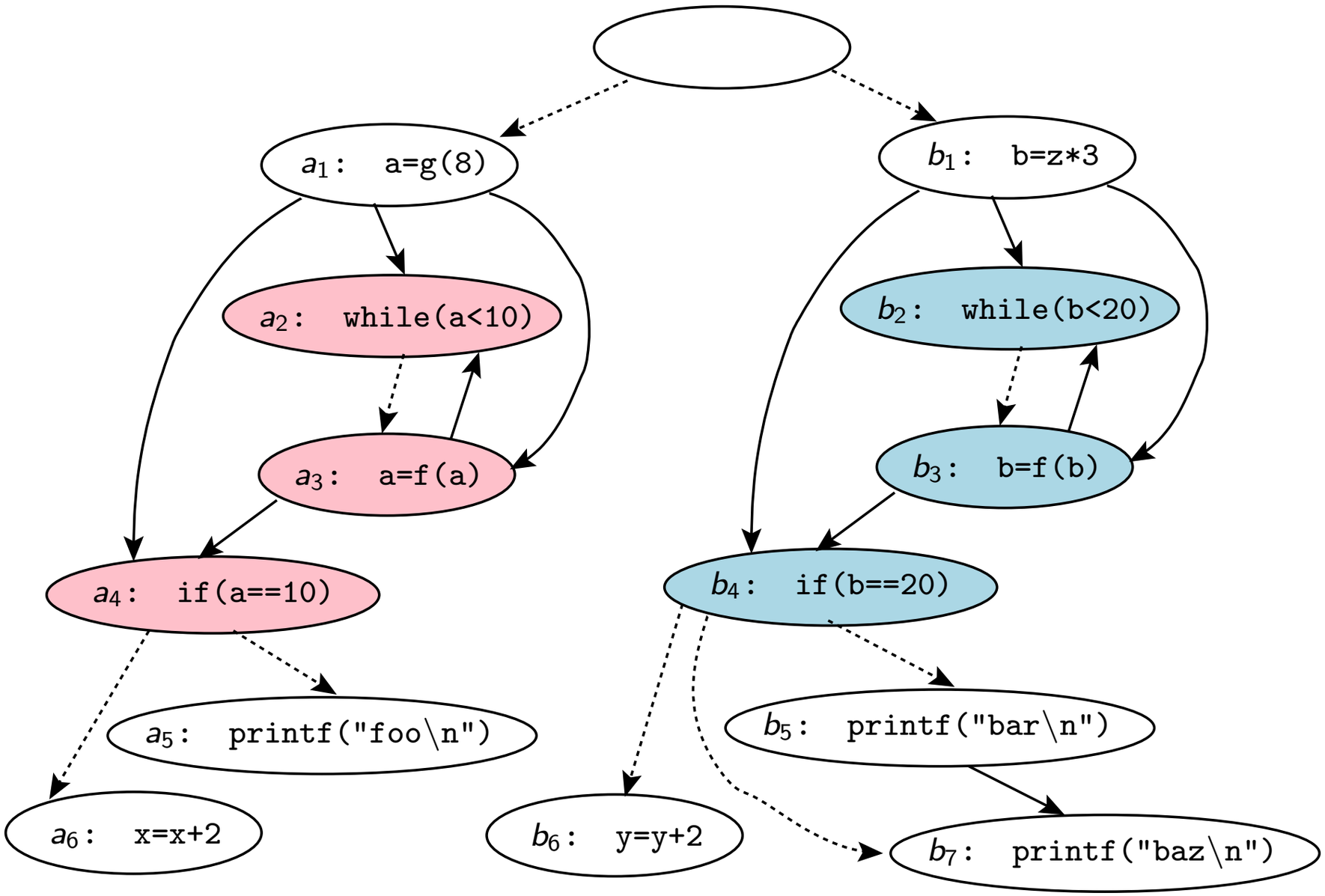
PDG-based clone detection



PDG-based clone detection

* `a4` and `b4` match. Add them to the slice.
* Consider `a4` and `b4`'s predecessors `a3` and `b3`
* `a3` and `b3` match too. Add them to the slice.
* Add `a2` and `b2` to the slice since they match and are predecessors of `a3` and `b3`

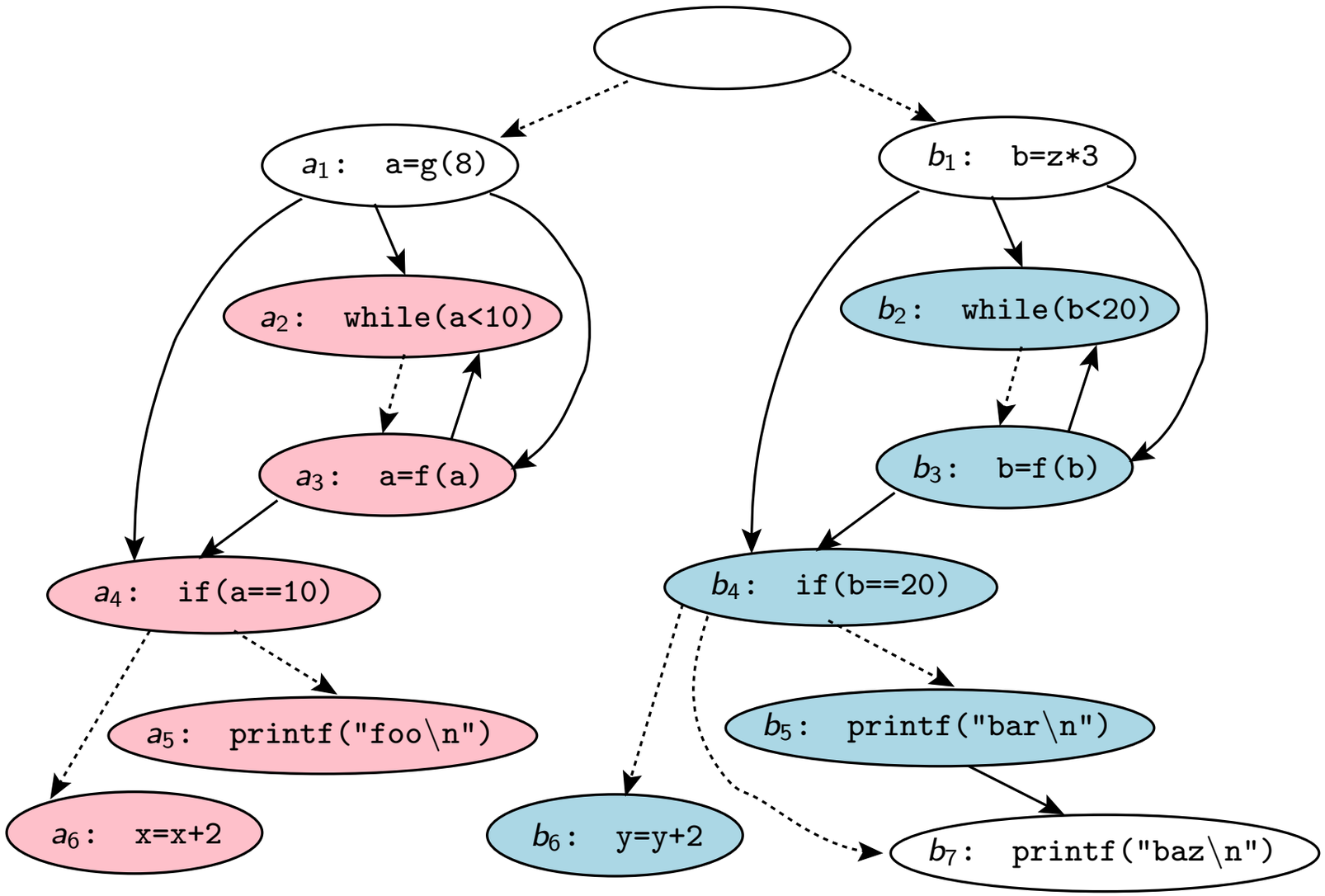
PDG-based clone detection



PDG-based clone detection

- `a5/b5` and `a6/b6` really should belong to the clone!
- But, backwards slice won't include them
- So, slice forward one step from any predicate in an `if` and `while` statements

PDG-based clone detection



PDG-based clone detection

- This algorithm handles
  - Clones where statements have been reordered
  - Clones that are non-continuos
  - And clones that have been intertwined with each other
- Depressing performance numbers. A 11.540 line C program takes 1 hour and 34 minutes to process

PDG-based plagiarism detection

- Uses PDGs, but for plagiarism detection
- Uses general-purpose subgraph isomorphism algorithm
- Uses a preprocessing step to weed out unlikely plagiarism candidates

PDG-based plagiarism detection

- What does it mean for one PDG to be considered a plagiarized version of another?
- We expect some manner of obfuscation of the code - equality is too strong!
- The two PDGs should be γ-isomorphic
- Set γ=0.9

PDG-based plagiarism detection

- Subgraph isomorphism testing is expensive - prune out 9/10 of all program pairs from consideration:
  - Ignore any graph which has too few nodes to be interesting
  - Remove (g,g') from consideration if $|g'| < \gamma|g|$ (would never pass a γ-isomorphism test)
  - Remove (g,g') if the frequency of their different node types are too different
    - For example if g consists solely of function call nodes and g' consists solely of nodes representing arithmetic operations then they are unlikely related

PDG summary

- A PDG is not affected by
  - Statement reordering
  - Variable renaming
  - Replacing while-loops by for-loops
  - Edge flipping
- The PDG is affected by:
  - Inlining and outlining
  - Add bogus dependencies to introduce spurious edges

Similarity Tools and Algorithms

BinDiff

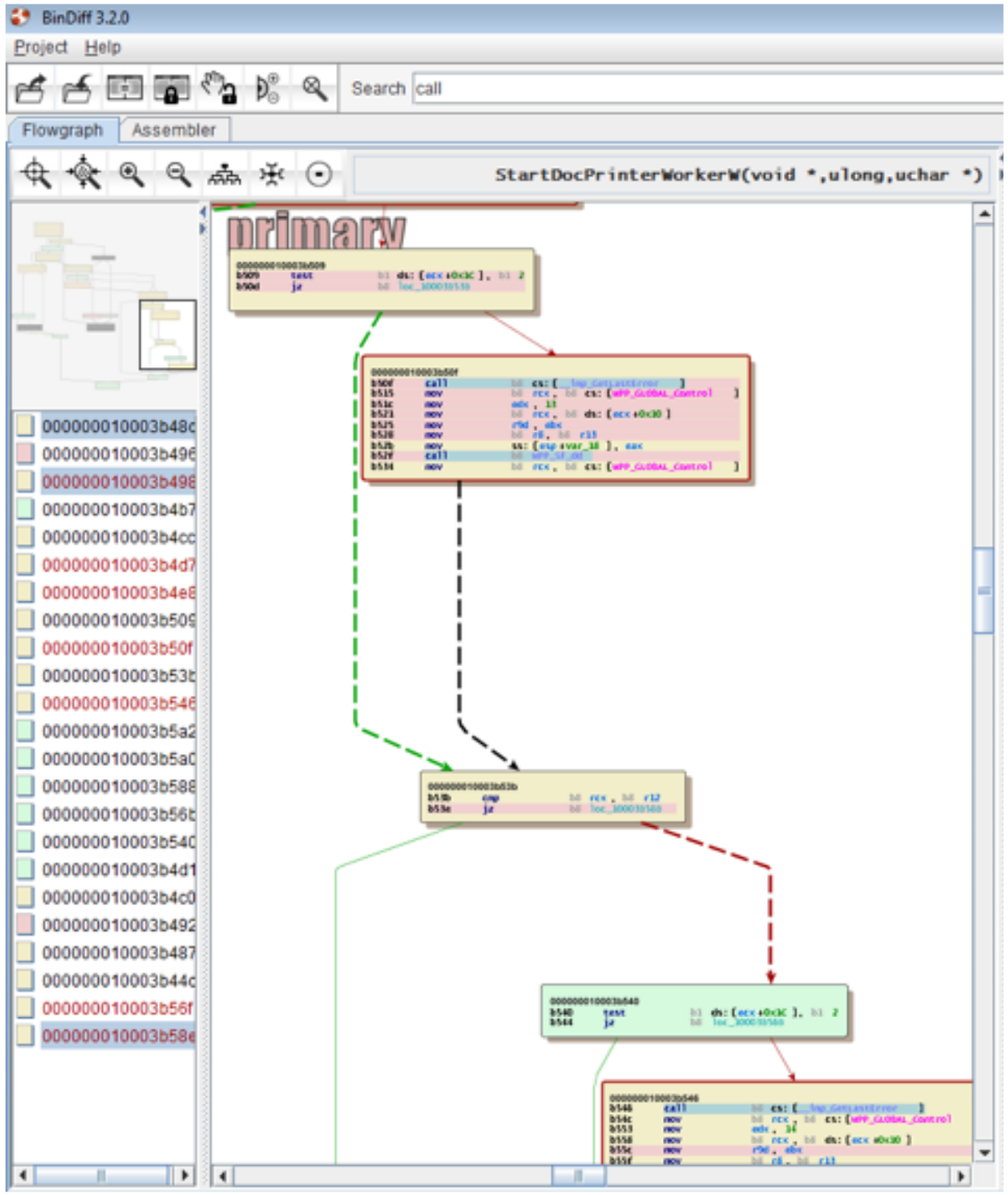
* BinDiff is a comparison tool for binary files, that assists vulnerability researchers and engineers to quickly find differences and similarities in disassembled code.
* BinDiff works on the abstract structure of an executables, ignoring the concrete assembly-level instructions in the disassembly
* BinDiff builds the flow graph of every function and the call-graph representing the relations between functions
* Matches functions wrt attributes (for example instruction permutation)
* When two functions matches it verifies if the functions called by that function match

zynamics BinDiff

Tool for comparing binaries

CFG + Call Graph manipulation

Match functions wrt attributes, recursively on the Call Graph

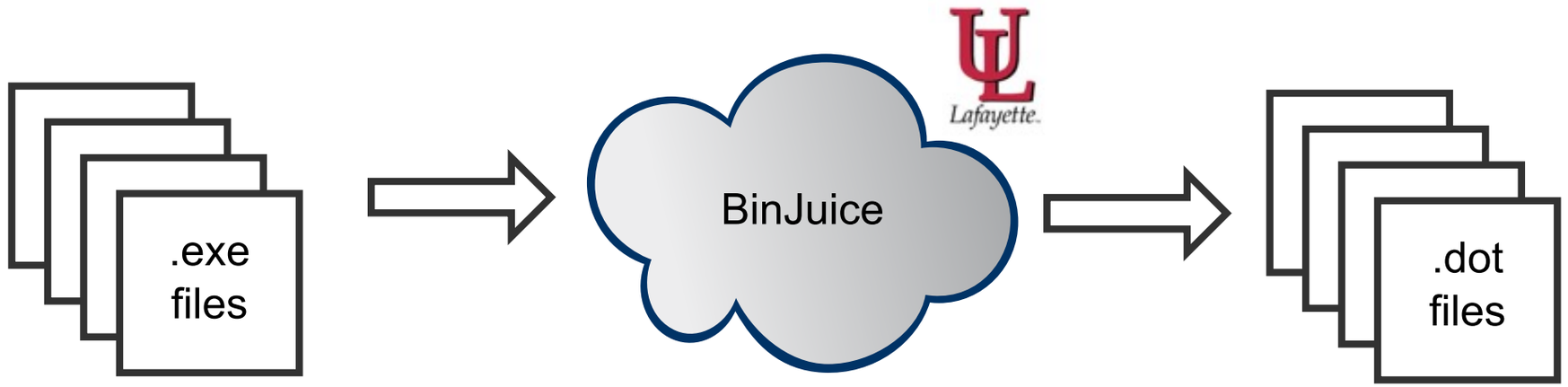


BinHunt

- BinHunt is a tool for finding *semantic differences* in binary programs by analyzing the control flow of the program.
- BinHunt constructs the CFG of each function and the call graph of the entire binary.
- BinHunt uses a *graph isomorphism algorithm* for finding the best match between functions and basic blocks (backtracking procedure that replaces bad matches with better ones).
- BinHunt uses *symbolic execution* and *theorem proving* for basic block matches.

BinHunt: automatically finding semantic differences in binary programs. Information and Communications Security. 2008

BinJuice



401290: b8 05 00 00 00 mov eax,0x5 401295: 81 c3 04 00 00 00 add ebx,0x4 40129b: 6b c3 imul eax,ebx	$\text{eax} = 5$ $\text{ebx} = (\text{def}(\text{ebx}) + 4) \times 5$ $= \text{def}(\text{ebx}) \times 5 + 20$	$A = \text{N1}$ $B = \text{def}(B) \times \text{N1} + \text{N2}$ where $\text{N2} = \text{N1} \times \text{N3}$ and $\text{type}(A) = \text{type}(B) = \text{reg32}$

(a) Code (b) Semantics (c) Juice

eax = 5 ebx = 10 mem(def(eax)) = def(ebx) mem(def(ebx)) = def(eax)	ebx = 10 ecx = 5 mem(def(ebx)) = def(ecx) mem(def(ecx)) = def(ebx)	R1 = N1 R2 = N2 mem(def(R1)) = def(R2) mem(def(R2)) = def(R1)	R = N mem(def(R)) = def(R)

(a) Semantics 1 (b) Semantics 2 (c) Juice (d) Abstracted Juice

BinJuice

```
push(ebp)
mov(ebp,esp)
sub(esp,16)
mov(eax,dptr(ebp))
push(edi)
push(ebx)
mov(ebx,ebp)
mov(edi,ebx)
pop(ebx)
push(ebx)
mov(ebx,46)
sub(edi,ebx)
pop(ebx)
mov(dptr(edi+42),eax)
pop(edi)
lea(eax,wptr(ebp-16))
push(esi)
mov(esi,1634038339)
mov(dptr(eax),esi)
pop(esi)
push(esi)
mov(esi,32509012)
add(esi,1733712160)
mov(dptr(eax+4),esi)
pop(esi)
push(edx)
mov(edx,4285804)
mov(dptr(eax+8),edx)
pop(edx)
push(eax)
```

(a) Variant 1

```
push(ebp)
mov(ebp,esp)
sub(esp,16)
mov(eax,dptr(ebp))
mov(dptr(ebp-4),eax)
lea(eax,wptr(ebp-16))
mov(dptr(eax),1634038339)
mov(dptr(eax + 4),1766221172)
mov(dptr(eax + 8),4285804)
push(eax)
```

(b) Variant 2

```
eax = -20 + def(esp)
ebp = -4 + def(esp)
esp = -24 + def(esp)
memdw(-24 + def(esp)) = -20 + def(esp)
memdw(-20 + def(esp)) = 1634038339
memdw(-16 + def(esp)) = 1766221172
memdw(-12 + def(esp)) = 4285804
memdw(-8 + def(esp)) = def(ebp)
memdw(-4 + def(esp)) = def(ebp)

Simplifications performed
 1766221172 = 32509012 + 1733712160
 -50 = -4 - 46
 -20 = -4 - 16
 -16 = -20 + 4
 -12 = -20 + 8
 -8 = -50 + 42
```

(c) Semantics

```
 A = -N1 + def(B)
 C = -N2 + def(B)
 B = -N3 + def(B)
 memdw(-N3 + def(B)) = -N1 + def(B)
 memdw(-N1 + def(B)) = N4
 memdw(-N5 + def(B)) = N6
 memdw(-N7 + def(B)) = N8
 memdw(-N9 + def(B)) = def(C)
 memdw(-N2 + def(B)) = def(C)
where
 N6 = N10 + N11
 -N12 = -N2 - N13
 -N1 = -N2 - N5
 -N5 = -N1 + N2
 -N7 = -N1 + N9
 -N9 = -N12 + N14
```

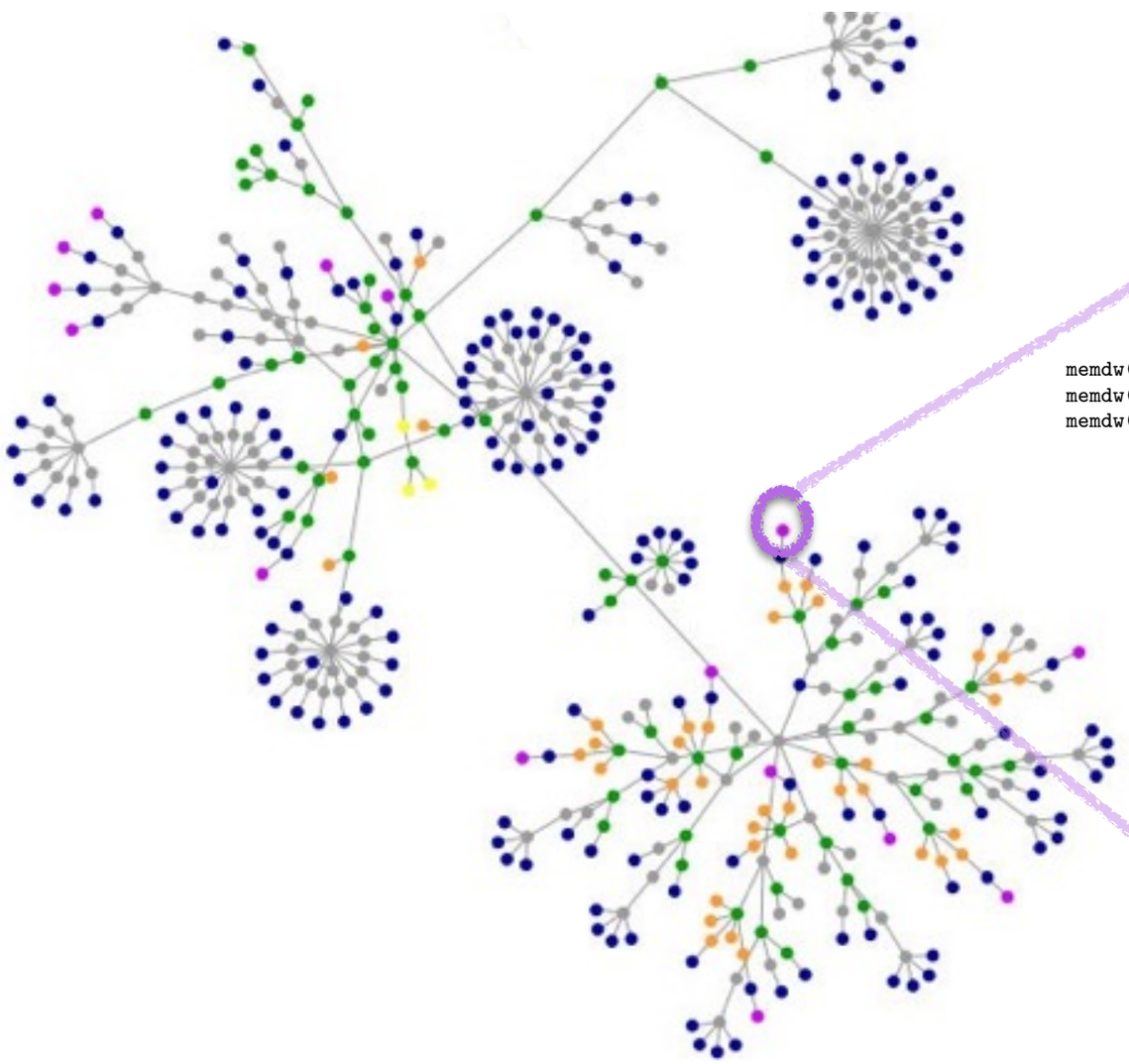
(d) Juice

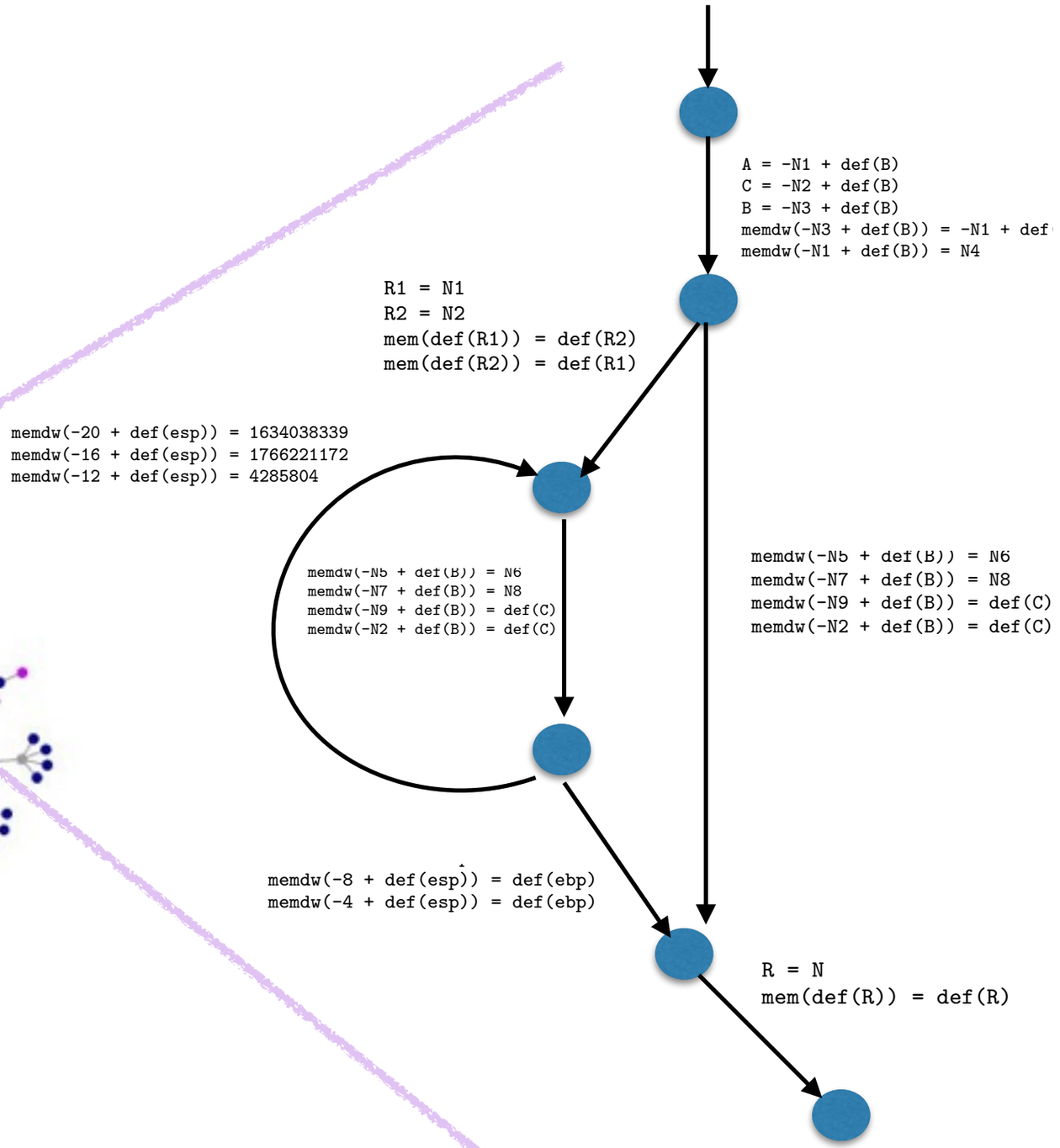
Symbolic semantics

Algebraic simplification

Fast location of similar code fragments using semantic juice PPREW 2013

BinJuice





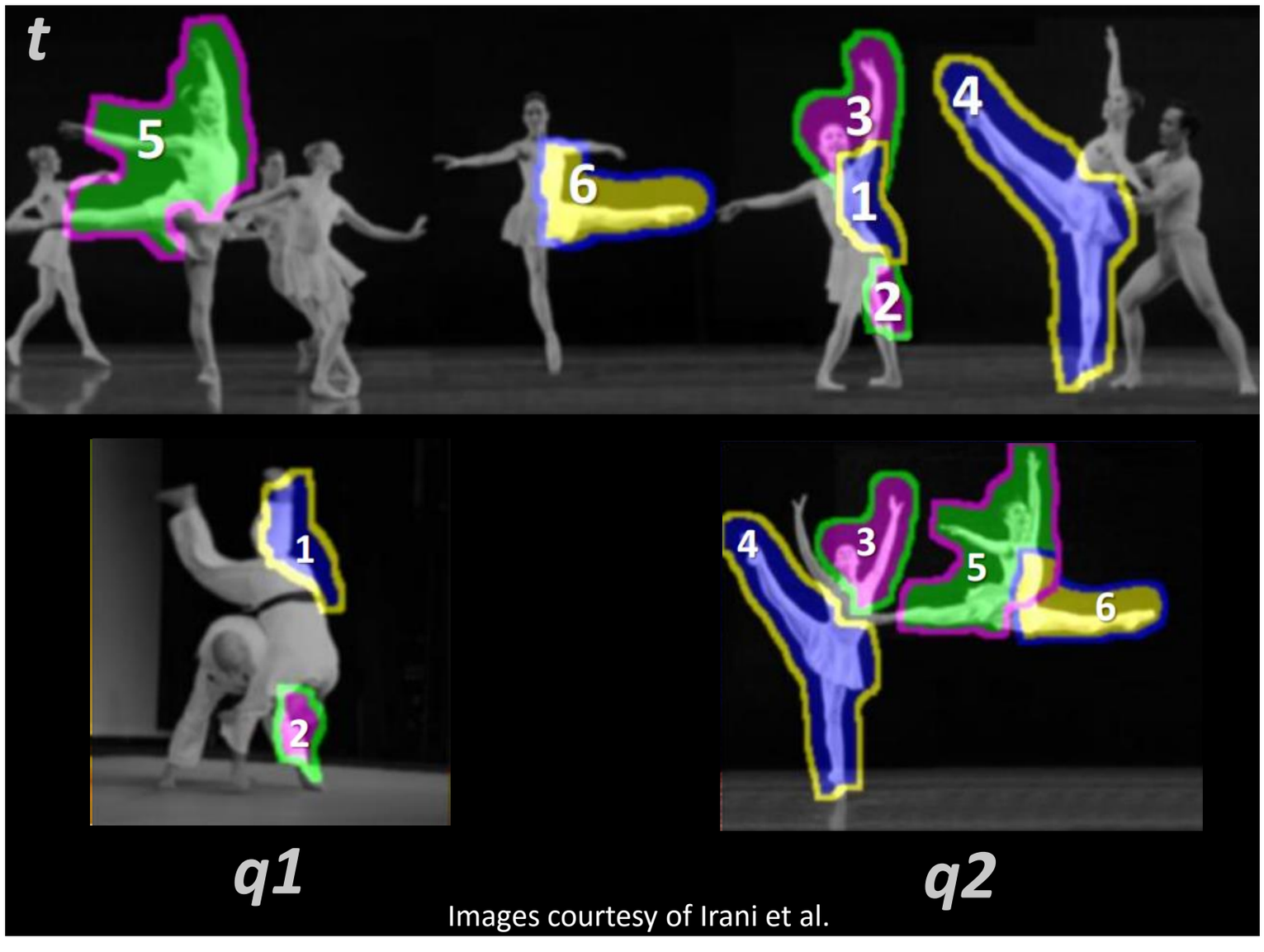
Fast location of similar code fragments using semantic juice PPREW 2013

Statistical Similarity of Binaries

- The problem: a security aware organization would like to use a *sample of the vulnerable product* (in binary form) to search for the vulnerability across all the software installed in the organization, where source code is mostly not available
- Given a query **q** and a large collection of target procedures **T** in binary form we want to *quantitatively* define the similarity of each procedure **t** in **T** to the query **q**
  - code compiled using different compiler versions
  - code compiled using different compiler vendors
  - different version of code was compiled (e.g. a patch)

Similarity by Composition

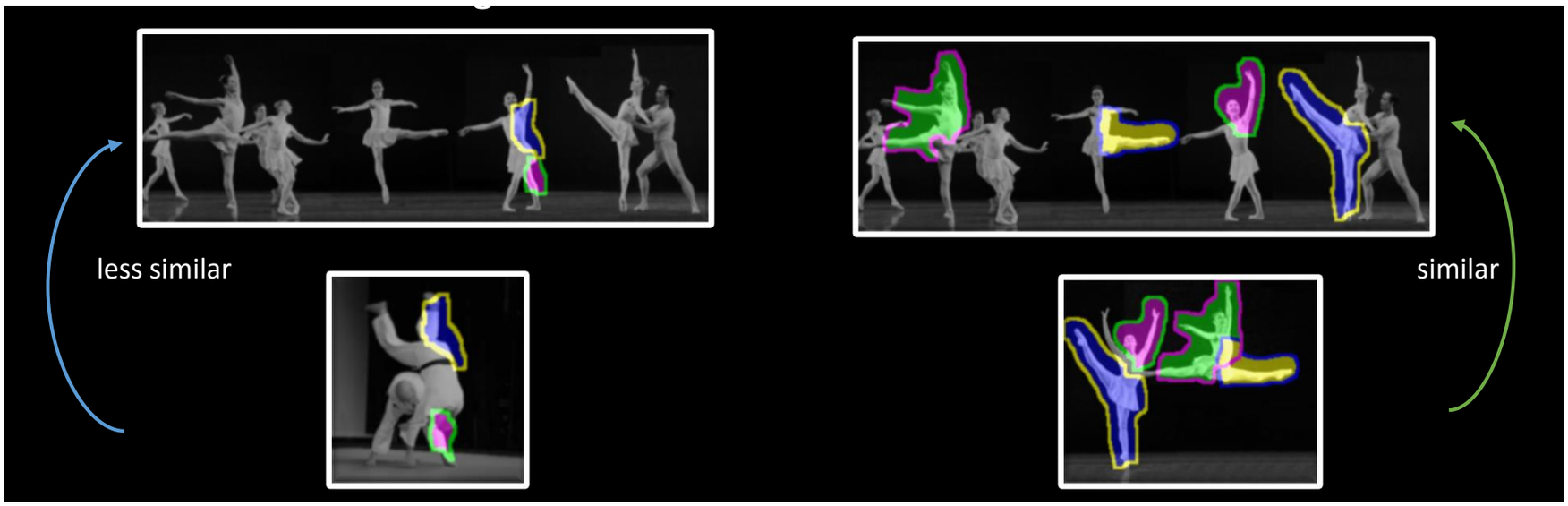
- IDEA: The main idea is to use *similarity by composition:* from image similarity where the key idea is that one image is similar to another if it can be composed using regions of the other image, and that this similarity can be quantified using *statistical* reasoning



Images courtesy of Irani et al.

Similarity by Composition

- **Irani et al 2006:**
  - image1 is similar to image 2 if you can compose image1 from the segments of image 2
  - segments can be transformed: rotated, scaled and translated
  - segments of statistical significance give more evidence (black background should be much less accounted for)



Similarity by Composition

- **IDEA**: decompose the code into smaller comparable fragments, define semantic similarity between fragments, and use statistical reasoning to lift fragment similarity into similarity between procedures

- *Procedures are similar if they share functionally equivalent, non trivial segments*

  - equivalent: allow for semantic preserving transformations (register allocation, instruction selection, etc…)

  - non-trivial: account for the statistical significance of each segment

Similarity by Composition

less similar

similar

```
①  shr      eax, 8
②  lea      r14d, [r12+13h]
③  mov      r13, rbx
③  lea      rcx, [r13+3]
   mov      [r13+1], al
   mov      [r13+2], r12b
   mov      rdi, rcx
```

t: Heartbleed, gcc v.4.9 -03

```
   mov      rsi, 14h
   mov      rdi, rcx
①  shr      eax, 8
   mov      ecx, r13
   add      esi, 1h
   xor      ebx, ebx
   test     eax, eax
   jl       short loc_22F4
```

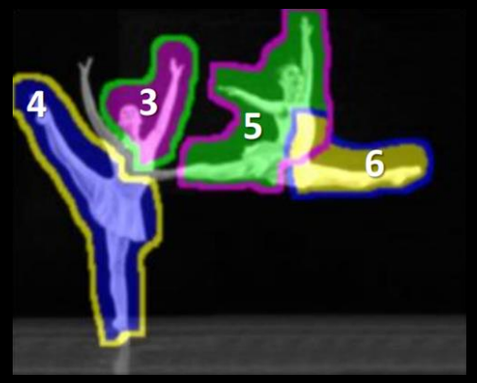
q_1: *Coreutils, gcc v.4.9 -03*

```
   mov      r9, 13h
③  mov      r12, rbx
   add      rbp, 3
   mov      rsi, rbp
③  lea      rdi, [r12+3]
   mov      [r12+2], bl
②  lea      r13d, [rcx+r9]
①  shr      eax, 8
```

q_2: *Heartbleed, clang v.3.5 -03*

Similarity by Composition for Binaries

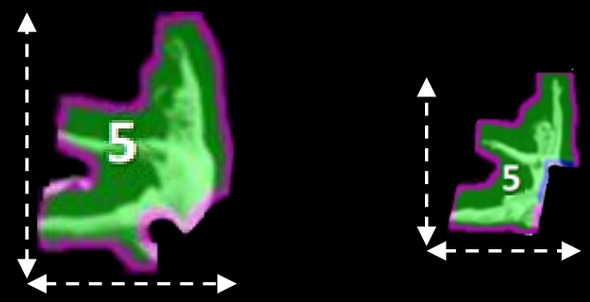
1. Decomposition




```
①  shr      eax, 8
②  lea      r14d, [r12+13h]
③  mov      r13, rbx
③  lea      rcx, [r13+3]
   mov      [r13+1], al
   mov      [r13+2], r12b
   mov      rdi, rcx
```

Heartbleed, gcc v.4.9 -03

2. Pairwise Semantic Similarity



Heartbleed, gcc v.4.9 -03

```
③  mov      r13, rbx
③  lea      rcx, [r13+3]
```

≈?

```
③  mov      r12, rbx
③  lea      rdi, [r12+3]
```

Heartbleed, clang v.3.5 -03

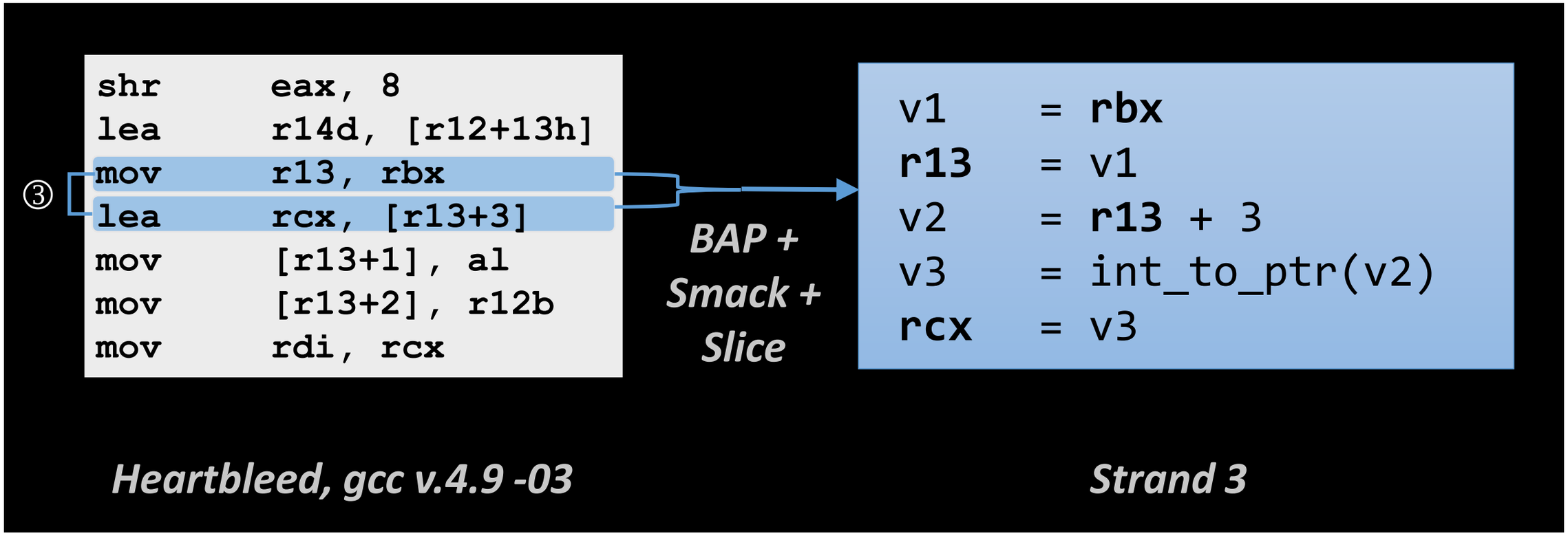
3. Statistical Similarity Evidence


```
①  shr      eax, 8
③  mov      r13, rbx
③  lea      rcx, [r13+3]
①  shr      eax, 8
③  mov      r12, rbx
③  lea      rdi, [r12+3]
①  shr      eax, 8
```

CORPUS

Procedure Decomposition

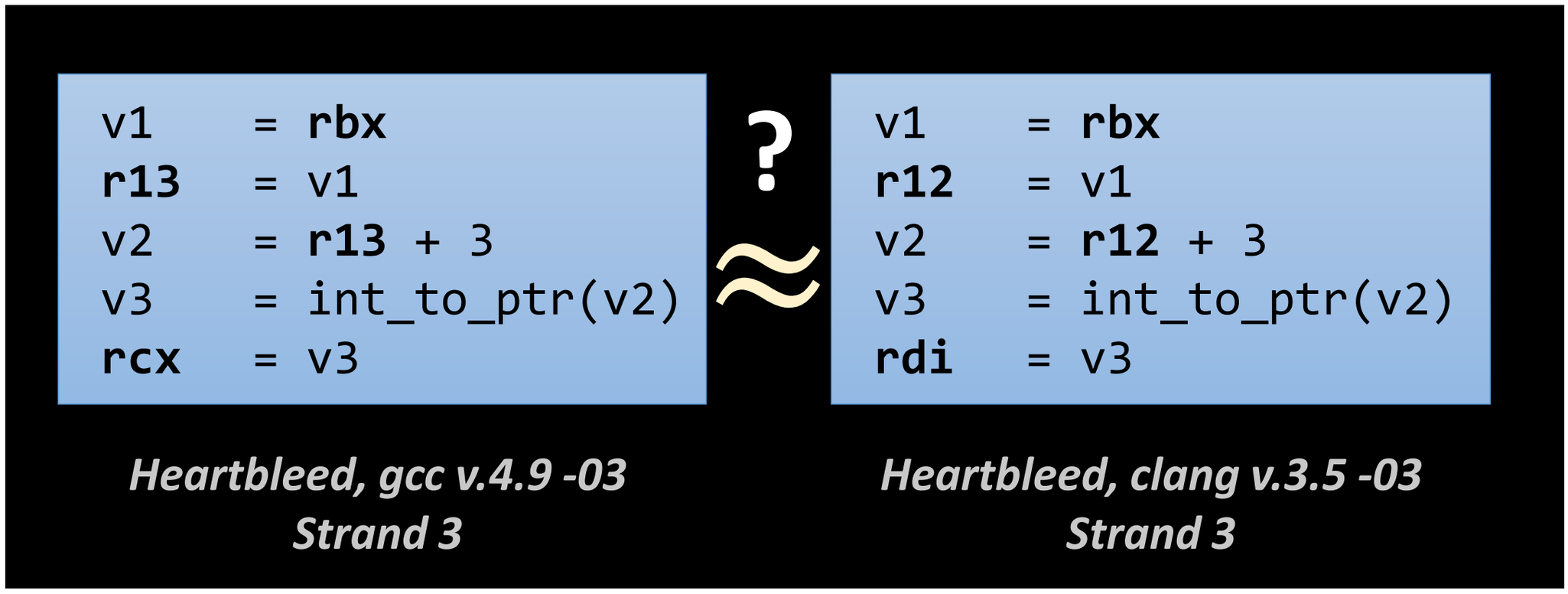
- Decompose the procedure into comparable units
- STRANDS: the set of instructions required to compiute a certain variable's value
- Get all strands by applying backward program slicing on the basic block level until all variables are covered (strands need not to be syntactically contiguous)



Procedure Decomposition

* Since we consider every basic block separately the inputs of a block are the registers and memory locations used in the block but defined outside the block
* The identification of the input and output variables of a block is important in the process of comparing strands

Pairwise Semantic Similarity



YES!

Pairwise Semantic Similarity

```
v1    = r12
v2    = 13h + v1
v3    = int_to_ptr(v2)
r14   = v3
v4    = 18h
rsi   = v4
v5    = v4 + v3
rax   = v5
```

Heartbleed, gcc v.4.9 -03
Strand 6

?

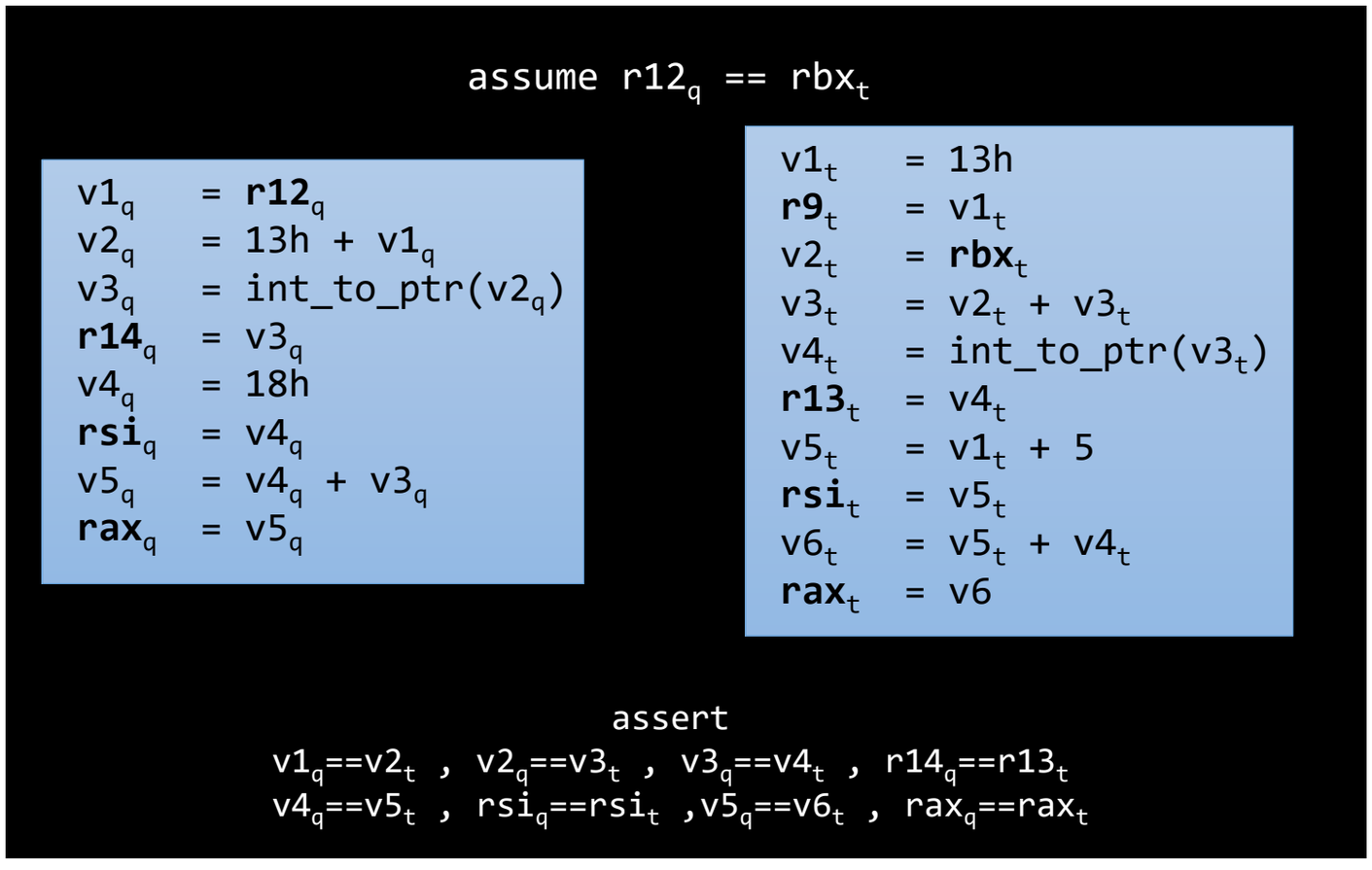
≈

```
v1    = 13h
r9    = v1
v2    = rbx
v3    = v2 + v3
v4    = int_to_ptr(v3)
r13   = v4
v5    = v1 + 5
rsi   = v5
v6    = v5 + v4
rax   = v6
```

Heartbleed, clang v.3.5 -03
Strand 11

???

Pairwise Semantic Similarity



- add equality assumptions over the inputs of the two strands
- add assertions that check the equality of all output variables (including temporaries)
- check the assertions using a *program verifier* and count how many variables are equivalent

Pairwise Semantic Similarity

assume $r12_q$ == rbx_t

$v1_q$ = **$r12_q$**
$v2_q$ = 13h + $v1_q$
$v3_q$ = int_to_ptr($v2_q$)
$r14_q$ = $v3_q$
$v4_q$ = 18h
rsi_q = $v4_q$
$v5_q$ = $v4_q$ + $v3_q$
rax_q = $v5_q$

$v1_t$ = 13h
$r9_t$ = $v1_t$
$v2_t$ = **rbx_t**
$v3_t$ = $v2_t$ + $v3_t$
$v4_t$ = int_to_ptr($v3_t$)
$r13_t$ = $v4_t$
$v5_t$ = $v1_t$ + 5
rsi_t = $v5_t$
$v6_t$ = $v5_t$ + $v4_t$
rax_t = v6

assert

$v1_q$==$v2_t$, $v2_q$==$v3_t$, $v3_q$==$v4_t$, $r14_q$==$r13_t$
$v4_q$==$v5_t$, rsi_q==rsi_t ,$v5_q$==$v6_t$, rax_q==rax_t

Quantify Semantic Similarity

- Use the percentage of variables from s_q that have an equivalent counterpart in s_t to define probabilities:
  - **VCP(s_q,s_t)** is the percentage of variables of **q** that have a equivalent counterpart in **t**
- it is an *asymmetric* measure
- **Pr(s_q | s_t)** is the probability that s_q is input/output equivalent to s_t

Pairwise Semantic Similarity

assume $r12_q$ == rbx_t

$v1_q$	=	**r12**$_q$
$v2_q$	=	13h + $v1_q$
$v3_q$	=	int_to_ptr($v2_q$)
r14$_q$	=	$v3_q$
$v4_q$	=	18h
rsi$_q$	=	$v4_q$
$v5_q$	=	$v4_q$ + $v3_q$
rax$_q$	=	$v5_q$

$v1_t$	=	13h
r9$_t$	=	$v1_t$
$v2_t$	=	**rbx**$_t$
$v3_t$	=	$v2_t$ + $v3_t$
$v4_t$	=	int_to_ptr($v3_t$)
r13$_t$	=	$v4_t$
$v5_t$	=	$v1_t$ + 5
rsi$_t$	=	$v5_t$
$v6_t$	=	$v5_t$ + $v4_t$
rax$_t$	=	v6

assert

$v1_q$==$v2_t$, $v2_q$==$v3_t$, $v3_q$==$v4_t$, $r14_q$==$r13_t$
$v4_q$==$v5_t$, rsi_q==rsi_t ,$v5_q$==$v6_t$, rax_q==rax_t

- **VCP(s_q,s_t) =1** since all the 8 variables form **q** have a counterpart in **t**
- **VCP(s_t,s_q) = 8/9**

Local Statistical Evidence

- Define a *Local Evidence Score* to quantify the statistical significance of matching each strand
  - **Pr(s_q | H_0)** is the probability of randomly finding a matching for s_q in the wild, it is given by the average value of **Pr(s_q | s_t)** over the entire corpus s_t in any possible target

$$LES(s_q|t) = \log \frac{\max_{s_t \in t} \Pr(s_q|s_t)}{\Pr(s_q|H_0)}$$

- **LES(s_q | t)** measures the level of confidence for s_q to have a non-trivial semantic equivalent strand in **t**

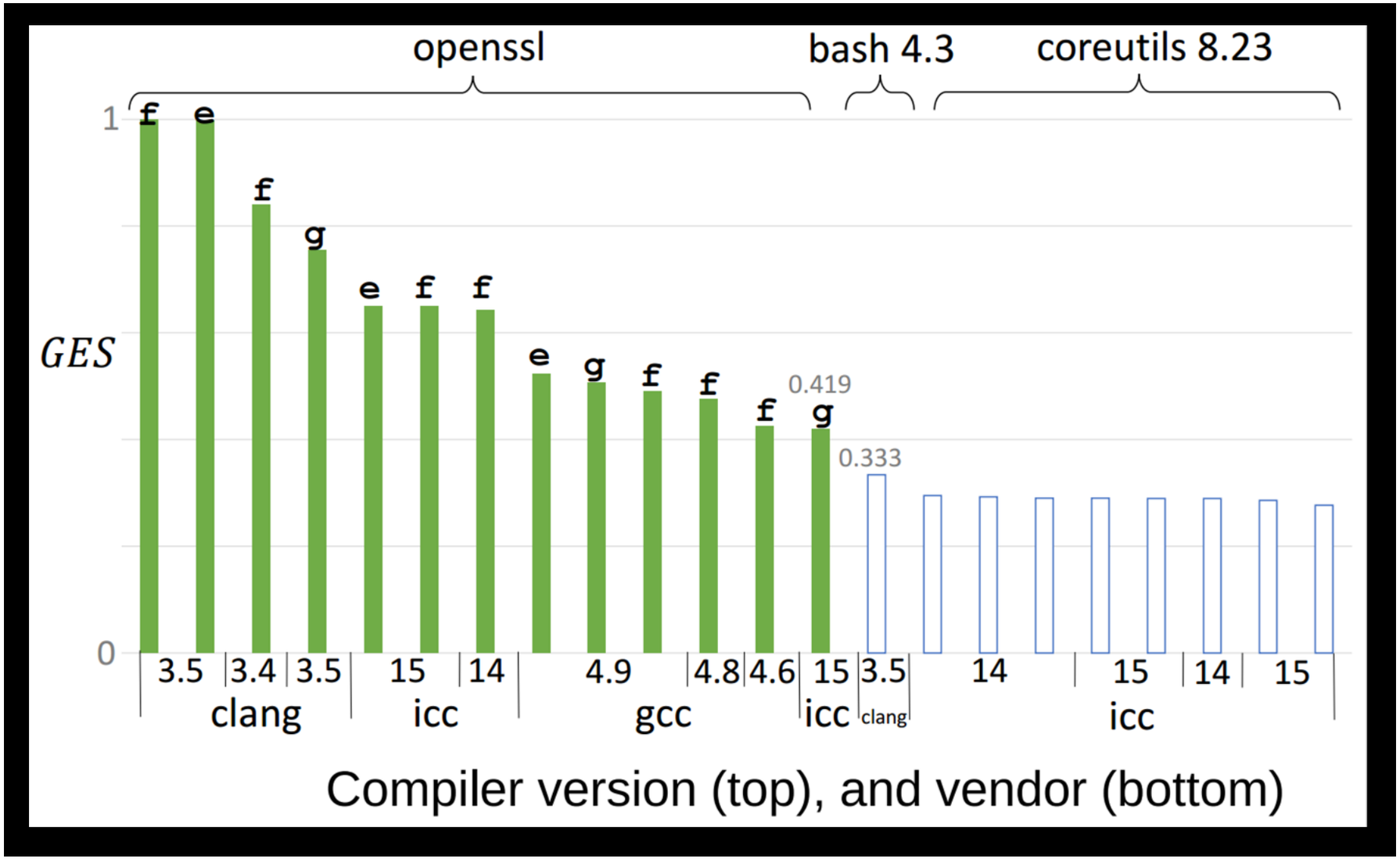
Global Statistical Evidence

- Define a *Global Evidence Score* represents the likelihood that a query procedure **q** is similar to a target procedure **t**
  - it is given by the sum of the likelihoods that each strand $\mathbf{s_q}$ in **q** has a similar strand $\mathbf{s_t}$ in **t**

$$GES(q|t) = \sum_{s_q \in q} LES(s_q|t)$$

Evaluation

Finding Heartbleed



Query: vulnerability heart bleed compiled with Clang
Improvement wrt previous work in terms of false positive rate and similarity detection (BinDiff, TRACY)

Android Similarity Analysis

Marketplaces

Smartphone apps can be bought and installed from centralized marketplaces:

* App Store (Apple)
* Play Store (Google)
* Third-party marketplaces such as Cydia, Amazon AppStore, proandroid, etc. These marketpalces contain:
  - Apps available also in the official marketplaces (reach more users)
  - Apps only available on third-party marketplaces that target specific customers (countries, languages)
  - Apps that are repackaged from the official marketplaces and re-distributed to third-party marketplaces

App-repackaging

- Repackaged apps are based on legitimate ones but include some "value-added" functionality or modification
  - repackaged with *additional malicious payloads* or exploits leads to compromised phones, increase of phone bills and theft of personal information
  - app developers: *intellectual property violated*, in-app revenues stolen, reputation impaired
  - marketplace: customer move to other marketplaces

It is important to identified repackaged apps

DroidMOSS

- **DroidMOSS**: similarity measurement system that applies *fuzzy hashing* to effectively localize and detect the changes from app-repackaging behavior. And use *edit distance* to measure the similarity

- Six third-party Android marketplaces:
  - Two from the unites states with 6.296 apps
  - Two from China with 12.595 apps
  - Two from East Europe with 4.015
- Apps collected in the first week of March 2011
- Compared to 68.187 apps collected from the official Android market in the same period of time
- **DroidMOSS** computes the similarity score of the 81.824.000 of pairs of apps

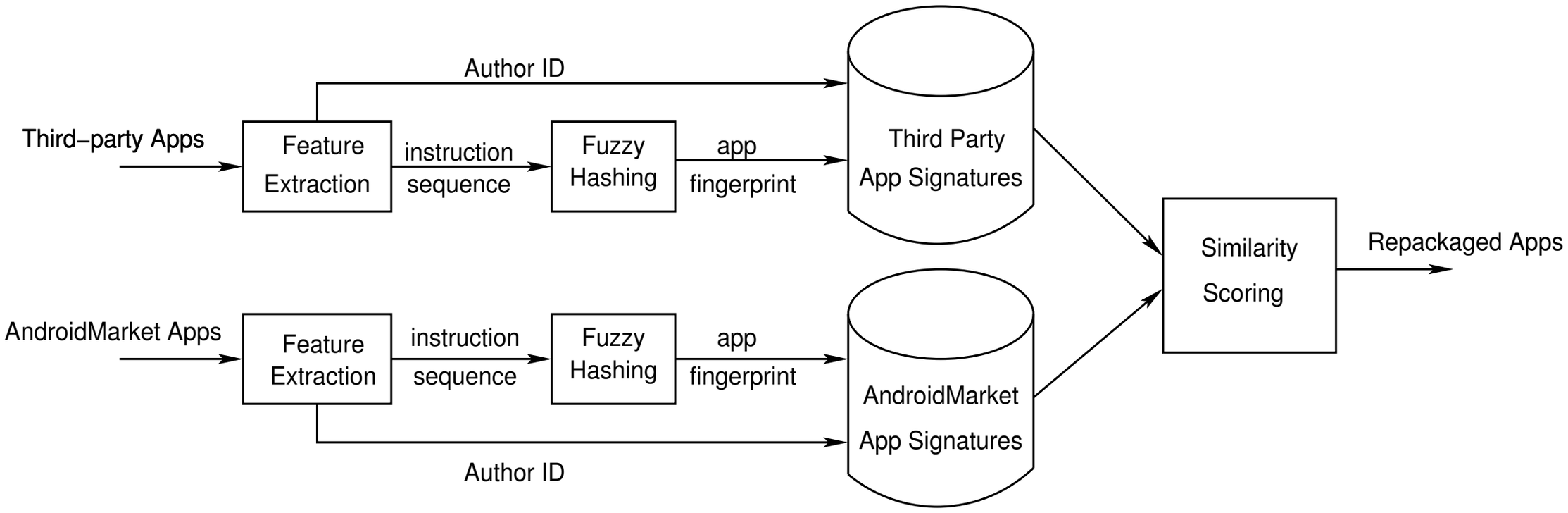
DroidMOSS

- **DroidMOSS**:
- 5% to 13% of apps hosted in these marker-places are re-packaged, with a false positive rate ranging form 7,1% to 13,3%.
- 13,5% to 30% of apps in these alternative marketplaces are simply redistributed from the official Android market
- **DroidMOSS** is design taking into account
  - accuracy
  - scalability
  - efficiency

DroidMOSS

- Assumptions:
  - Consider only Java code and not native code (harder for repackager to modify)
  - The signing keys from app developers are not leaked (it is not possible that a repackaged app will be signed by the same author as the original one)
- Repackaged apps are similar and are signed with different developers keys

DroidMOSS Overview



- Extract forum each app the instructions and the author information
- Generate the app fingerprint
- Compare signatures and generate similarity scores

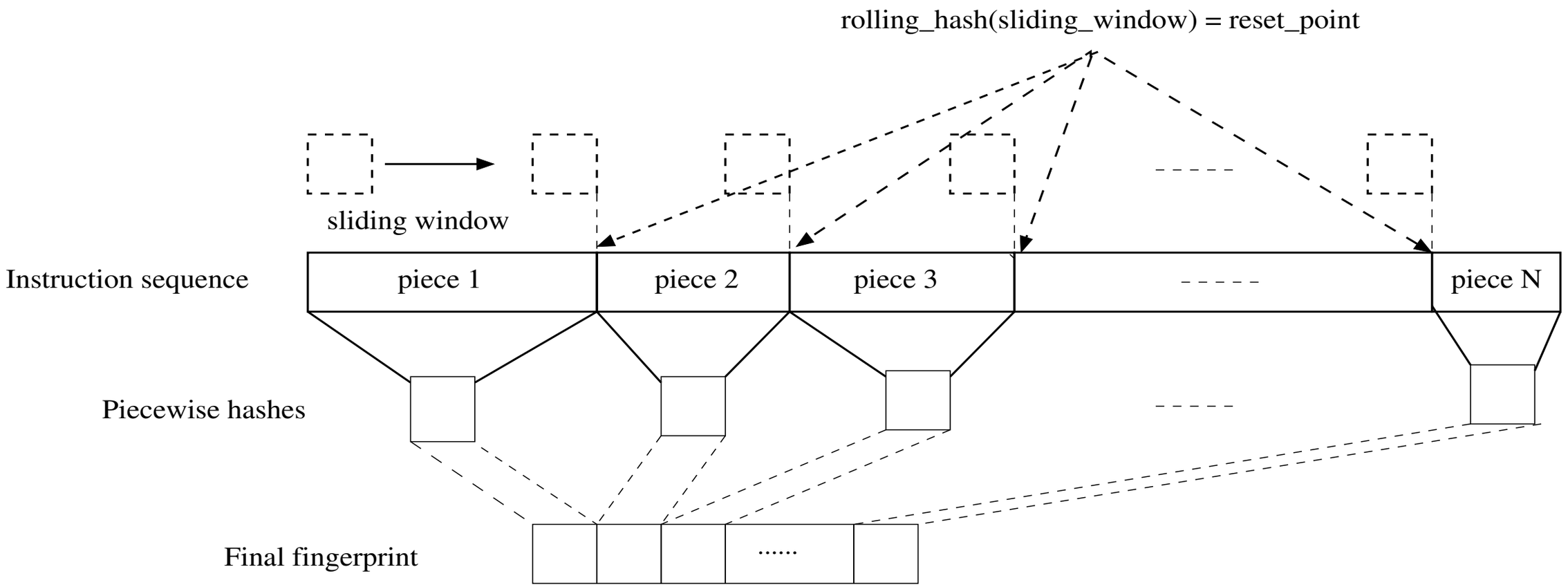
Feature Extraction

- APK file is a compressed file that contains:
  - **class.dex** file that contains the Dalvik bytecode for execution:
    - They use disassembler *baksmali* and retain only the opcodes (easy for repackagers to modify the name of operands and hard to change the actual instructions).
    - Moreover, they use a white-list of ad SDK libraries commonly used to display ads to remove them from the extracted code. These libraries introduce useless noise.
  - **META-INF** subdirectory contains the *author information.* In particular it contains the full developer certificate from which it is possible to extract the developer name, contact and organization information and the public key fingerprint. The certificate is mapped to an unique 32-bit identifier.

Fingerprint Generation

- Compute the hash of the application code allows to recognize identical apps but not similar apps
- **Fuzzy hashing**: the instruction sequence is condensed into a much shorter fingerprint that is then used to measure the similarity
  - The reduction into shorter fingerprints should minimize the change to the similarity of the two sequences
  - The instruction sequence is divided into pieces that constitute an independent unit: if the repackage process changes one piece its impact on the final fingerprint is localized and contained within this piece
  - Challenge: fix the boundary of the code pieces
  - **DoridMOSS** uses a *sliding window* that starts at the beginning of the instruction sequence and moves forward until its rolling hashing value equals a preselected *reset point* which determines the boundary of the current piece

Fuzzy hashing for fingerprint generation



Similarity Scoring

- The similarity measures the *edit distance* of the so derived fingerprints, not on the long instruction sequences
- Fix a threshold for deciding when two apps are considered similar
- Fuzzy hashing allows to localize the chased made by repackaged apps

Similarity Scoring

Marketplace	Total Number of Apps
US1 (slideme)	3108 (29.8%†)
US2 (freewarelovers)	3188 (13.2%†)
CN1 (eoemarket)	8261 (30%†)
CN2 (goapk)	4334 (13.5%†)
EE1 (softportal)	2305 (19.6%†)
EE2 (proandroid)	1710 (20.2%†)
Official Android Market	68187

- For each alternative marketplace, we report the percentage of apps that are hosted in it but also have an identical copy in the official Android Market: to meet more users

Similarity Scoring

Third-party Marketplace	# Repackaged Apps Apps from DroidMOSS	# Repackaged Apps from Manual Analysis	Percentage
US1	24	22	11%
US2	13	12	6%
EE1	11	10	5%
EE2	15	13	6.5%
CN1	27	25	12.5%
CN2	28	26	13%

- Results obtained by randomly choosing 200 apps from each third-party marketplace and then detecting whether they are repackaged from some official Android market apps.

Comparison of the app manifest file from the original and repackaged app

Original *Angry Birds* (in the official Android Market)	Repackaged *Angry Birds* (in a US alternative marketplace)
<manifest android:versionCode="142" android:versionName="1.4.2" android:installLocation="preferExternal" package="com.rovio.angrybirds" xmlns:android="....."> <application android:label=.....> ..... <meta-data android:name="ADMOB_PUBLISHER_ID" android:value="**a14c9c5b4602e23**" /> <meta-data android:name="ADMOB_INTERSTITIAL _PUBLISHER_ID" android:value="**a14ca2471ee0891**" /> ..... </application> </manifest>	<manifest android:versionCode="142" android:versionName="1.4.2" android:installLocation="preferExternal" package="com.rovio.angrybirds" xmlns:android="....."> <application android:label=.....> ..... <meta-data android:name="ADMOB_PUBLISHER_ID" android:value="**a14ce0cb83321d2**" /> <meta-data android:name="ADMOB_INTERSTITIAL _PUBLISHER_ID" android:value="**a14ce0cbd3cc9a1**" /> ..... </application> </manifest>

- Example of the popular repackaged app Angry Brids where the vendor has embedded Admob (ad SDK) to collect the revenues
- The repackaged app did not modify any code in the original app, instead it modifies the Admob-specific identifier to redirect the revenues

The manifest file of a repackaged App

(the last receiver and service do not exist in the original APP)

```
. . . . . ..
<uses-permission android:name="android.permission.READ_SMS" />
<uses-permission android:name="android.permission.SEND_SMS" />
<uses-permission android:name="com.android.browser.permission.READ_HISTORY_BOOKMARKS" />
<uses-permission android:name="com.android.browser.permission.WRITE_HISTORY_BOOKMARKS" />
. . . . . ..
<receiver android:name="com.android.AndroidActionReceiver">
    <intent-filter>
        <action android:name="android.intent.action.SIG_STR" />
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>
. . . . . ..
<service android:name="com.android.MainService" android:process=":remote" />
```

- Trojanized version requests more permissions
- New functionality that when the system finishes booting launches a background service named com.android.MainService that fetches and executes instructions from a remote server, tuning the compromised device into a bot

StaDynA

- Static analysis fo Android applications can be hindered by the presence of *dynamic code updates* techniques:
  - Dynamic class loading
  - Reflection
- Malware use these techniques to conceal their malicious behaviors to static analyzers
- Existing static analyzers assume that the code *does not change dynamically* and that targets of reflection calls can be discovered in advance
- This is a *simplification* of what happens in the real world
- *Off-line instrumentation* is not a solution since it breaks the application signature and some malicious app check if the application signature corresponds to some hardcoded value and it not they may refuse to work
- Idea: *combine static and dynamic analysis* to reveal the hidden/updated behavior

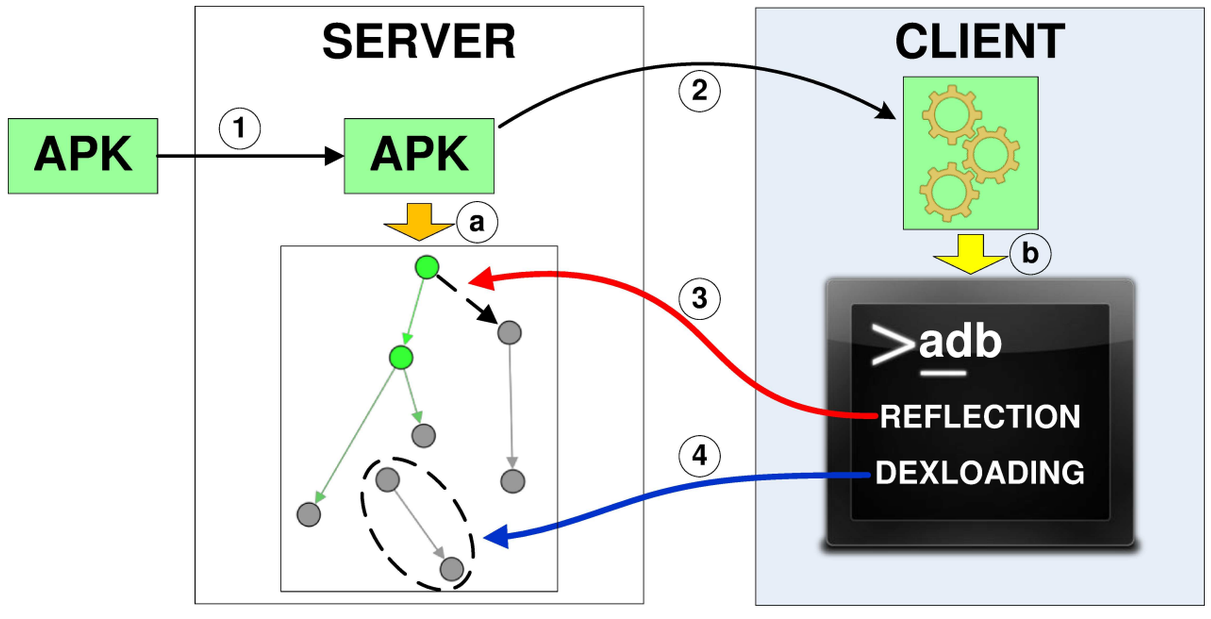
StaDynA

2013 Third party markets

Markets	Total Apps	DCL used by Apps	DCL used by %	Refl. used by Apps	Refl. used by %
Google Play	13863	2573	18.5%	12233	88.2%
Androidbest	1655	35	2.1%	1088	65.7%
Androiddrawer	2677	379	14.1%	2596	96.9%
Androidlife	1677	117	6.9%	1368	81.5%
Anruan	4230	162	3.8%	2868	67.8%
Appsapk	2664	112	4.2%	1907	71.5%
F-droid	1380	11	0.07%	792	52.8%
Malware	1260	251	19.9%	1025	81.3%
Total	**29406**	**3640**	**12.3%**	**23877**	**81.1%**

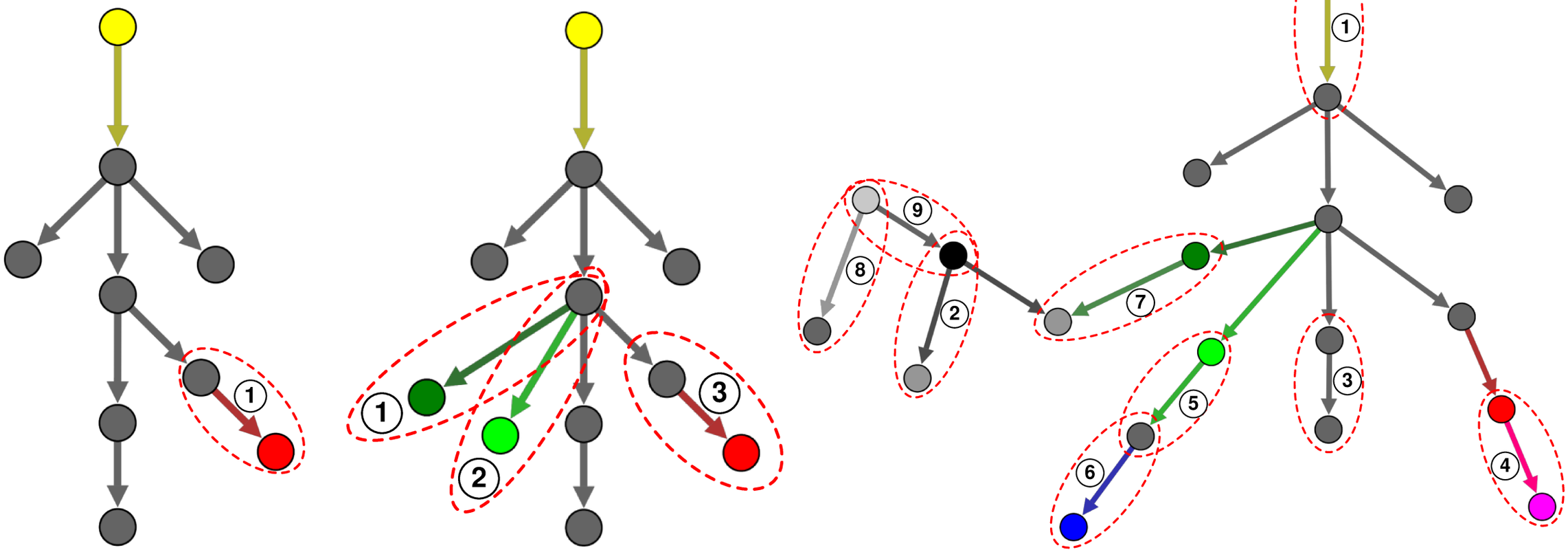
* Reflection: all the calls that *modify the method call graph* of the application (method invocation and object creation function)
* Reflection and dynamic class loading can be used to *extend the functionality of an application*. For example the app can provide stubs that process events using pieces of code (plug-in) written by different developers

StaDynA



- Server: the static analysis of the application is performed on the server (it is easy to plug-in any static analyzer).
  - Builds the method call graph (MCG) of the app and integrates the results of dynamic analysis coming from the client
- Client: it is a modified Android operating system, hosted either on a real device or on an emulator. The client runs the application when dynamic analysis is required

Method Call Graph

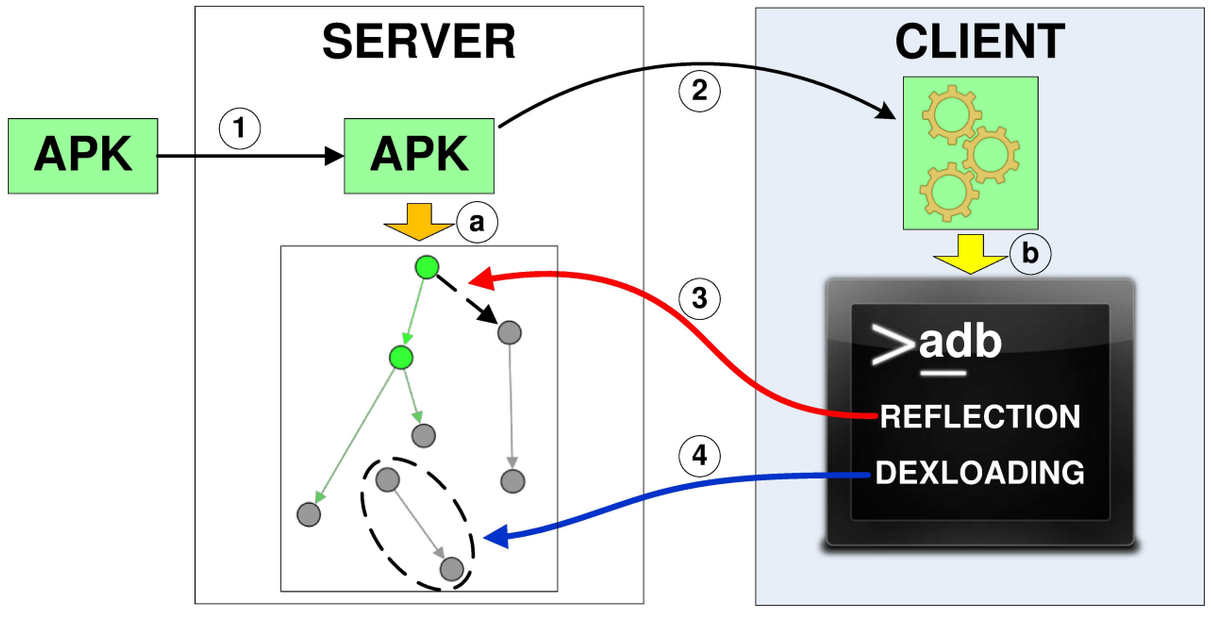


(a) (b) (c)

Figure 2: MCG of *demo_app* Obtained with a) AndroGuard b) StaDynA after Preliminary Analysis c) StaDynA after Dynamic Analysis Phase

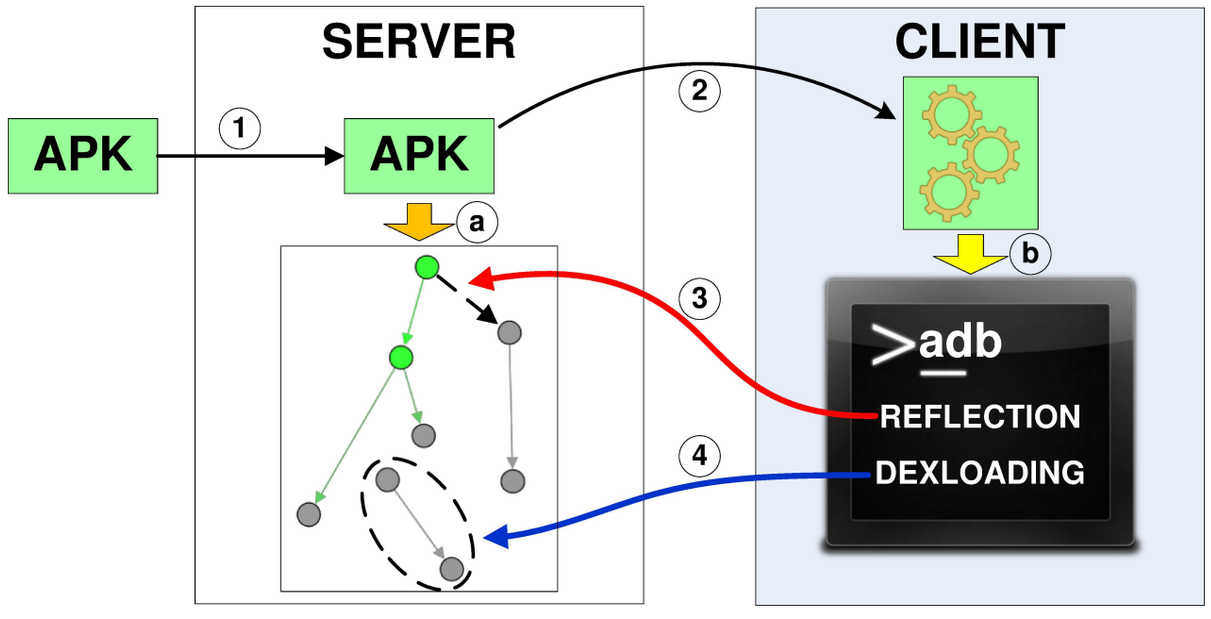
* Method Call Graph - MCG: identifies the caller-callee relationship for program methods
  * Detect malware
  * Identify potential privacy leaks
  * Find vulnerabilities

StaDynA: static analysis



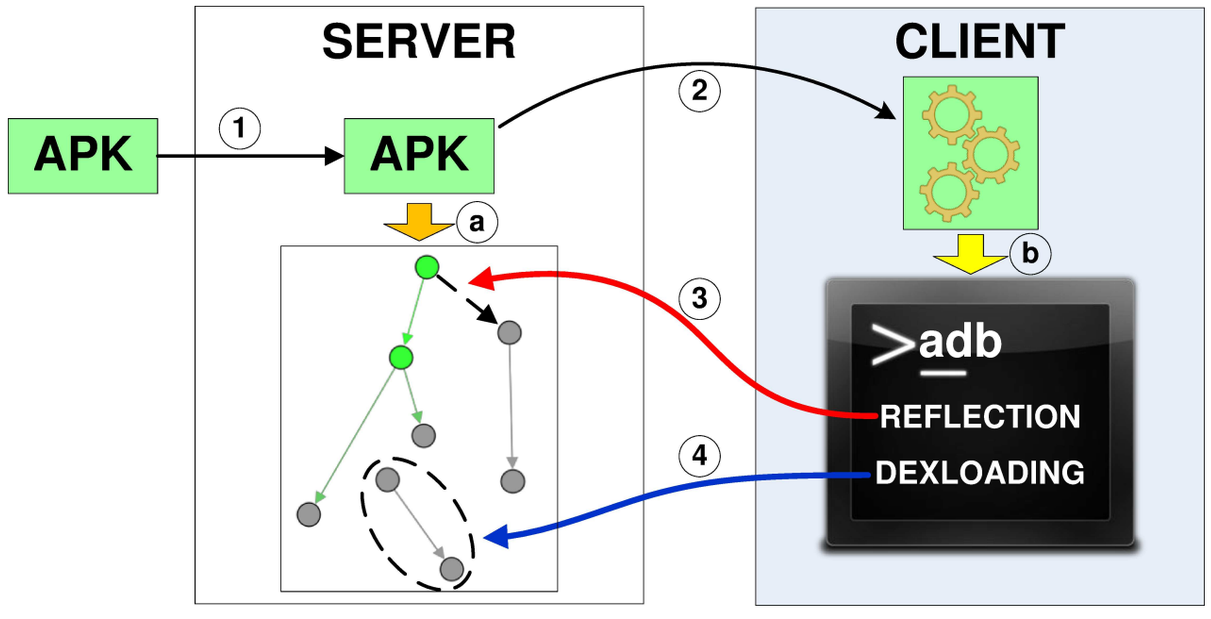
- Build the MCG of the application
- Dynamically loaded code cannot be analyzed and the corresponding nodes and edges are not present in the MCG
- The names of methods called through reflection may not be inferred if they are represented as encrypted strings or generated dynamically
- Detects the point in the MCG where the functionality of an application may be extended at runtime: use of specific API calls
- MOI - methods of interest are the ones that use DCL or reflection

StaDynA: dynamic analysis



- Dynamic analysis is performed if there are MOI in order to complement the MCG
- The dynamic analysis is performed on a modified Android OS that logs all events when the app executes a call using reflection or when additional code is loaded dynamically
- These information collected by the client is passed to the server and used to complete the MCG

StaDynA



- The execution of static and dynamic analysis are *interleaved*
- When we have DCL the dynamic analysis identifies the file used to get the code.
- The server downloads the file and performs static analysis on it
- StaDynA allows nested MOIs

Experimental Evaluation

Table 3: Evaluation: Number of MOIs for Each Operation

Apps	Refl. Invoke			Refl. NewInstance			DCL		
	Init.	Final	Triggered	Init.	Final	Triggered	Init.	Final	Triggered
Benign Applications									
FlappyBird	10	11	6	6	6	0	1	1	1
Norton AV	18	137	5	8	12	2	4	4	2
Avast AV	42	42	6	19	19	5	1	1	1
Viber	101	107	26	21	47	14	2	2	1
ImageView	6	6	5	2	2	2	0	0	0
Malicious Applications									
FakeNotify.B	68	68	68	9	9	9	0	0	0
AnserverBot	4	4	1	4	5	2	5	6	3
BaseBridge	5	5	1	2	3	2	2	3	3
DroidKungFu4	9	13	1	4	6	0	1	1	1
SMSSend	193	193	128	1	1	1	0	0	0

- **`Final`** is generally bigger than **`Initial`** since StaDynA analyzes also the dynamically loaded code
- **`Triggered`** represents the numbers of MOIs they were able to trigger during dynamic analysis
- The ration **`Triggered/Final`** can be seen as a measure of coverage: TO BE IMPROVED

Experimental Evaluation

Table 4: Evaluation: MCG Expansion

Apps	Nodes		Edges		Perm. Nodes	
	Init.	Final	Init.	Final	Init.	Final
Benign Applications						
FlappyBird	8592	8614	11014	11031	9	9
Norton AV	42886	55372	65960	85665	63	81
Avast AV	31317	32363	43554	44956	22	25
Viber	42536	46312	60078	65627	67	71
ImageView	5708	5713	6488	6496	7	7
Malicious Applications						
FakeNotify.B	148	171	137	191	1	2
AnserverBot	1006	1614	1138	2093	12	23
BaseBridge	1172	1780	1364	2333	14	25
DroidKungFu4	1550	21168	1779	23589	26	250
SMSSend	431	537	826	951	0	3

- `Perm. Nodes` shows the number of detected API methods protected with dangerous permissions.
- In malware many API that require dangerous permissions are loaded dynamically