THREATRAY

Using systematic code reuse analysis to create robust YARA rules

www.threatray.com

Agenda

- Introduction and goals
- YARA rules based on code
- Code search engine: Finding code reuse at scale
- Exercises: Building a YARA rule creation pipeline

Speakers

- Carlos Rubio Ricote
- David Pastor Sanz
- Jonas Wagner

How to get started with the hands-on exercises

- The ZIP file from the email contains everything you need for the workshop.

- You need: Linux (ideally Ubuntu) environment and a Docker (+docker-compose) installation.
  - We're working with malware, so ideally choose an isolated VM.
  - Minimum specs: 4 vCPUs, 8 GB RAM.

- Unzip the ZIP file and open a terminal into the folder extracted. If you don't have docker installed, then run `bash install.sh -i -p` (it will install and prepare docker for you).

- Run `bash install.sh -d` go to your browser and open the following page: http://0.0.0.0:9999/notebooks/Indices.ipynb

Introduction and goals

YARA

- YARA is a standard for detection and identification of malware attacks.

- “Easy to learn, hard to master”, needs expert knowledge and possibly time-consuming validation.

- Often done manually, but lots of opportunities to automate or support the process.

- Roughly two types of rules, based on text strings or based on bytes.

  - Currently, most publicly available rules are majorily composed by (text) strings:
    - Rule sets: Mike Worth [1], Florian Roth [2], YaraRules [3], deadbits [4], [redacted], …
      - Files: 2,516, Rules: 26,515
      - 73,295 (75.25%) text strings, 23,367 (23,99%) bytes, 736 (0,76%) regex

https://www.botconf.eu/wp-content/uploads/2019/12/B2019-Bilstein-Plohmann-YaraSignator.pdf

Goals of this workshop

- Get some background and motivation on YARA rules based on code.
- Understand how to get from binary code to a YARA rule.
- Understand how code search engines work.
- Build an automated YARA rule creation pipeline with code search engines and YARA rule creation tooling.
- Use the pipeline to automatically create code-based YARA rules for a set of malware families.

Hands-on exercises during the workshop

- Get to know code search engine using **Binlex.**
- Get to know code search engine using **FunctionSimSearch.**
- Building the pipeline using Binlex, FunctionSimSearch and **mkYARA.**
- Use the pipeline and create rules for malware families.

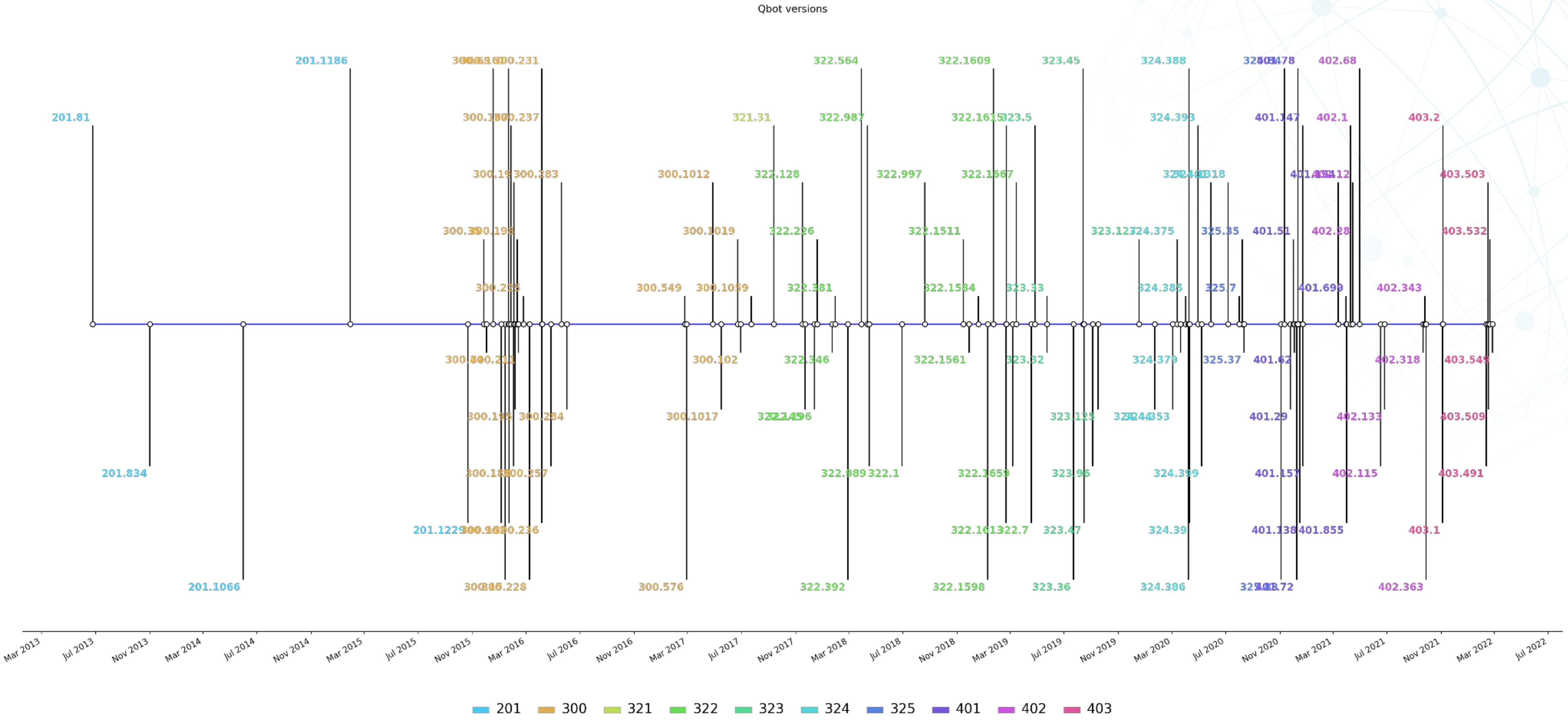
THREATRAY

YARA rules based on code

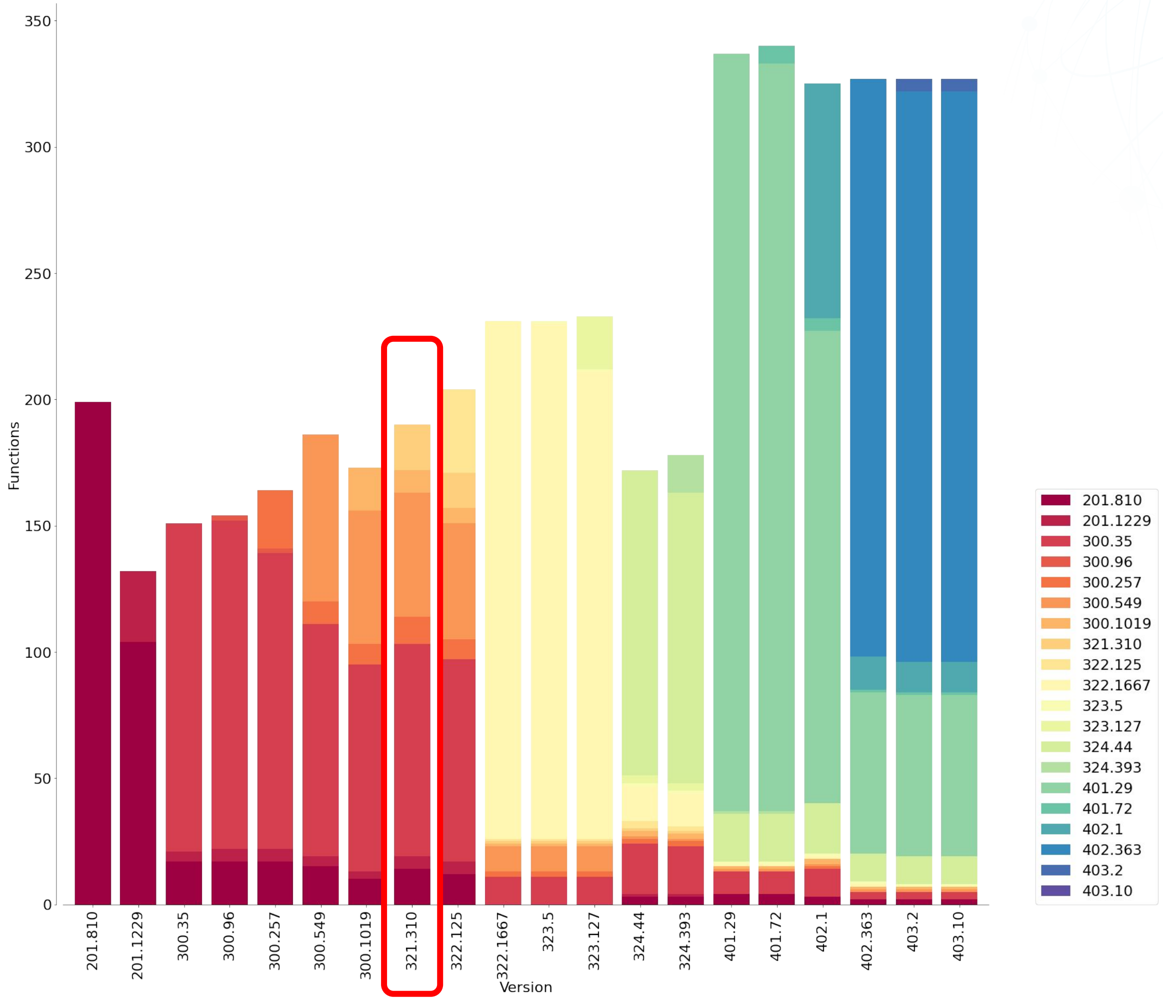
Code-based YARA rules (for identification)

- Robustness and longevity of code
- Uniqueness
- Automation and pre-validation
- But…

Longevity of code - Qbot



Longevity of code - Qbot



THREATRAY

What makes a good code-based rule?

- **Unique code**: Selected code is unique / identifying for a family. Exclude goodware code.
- **Normalized code**: Independent of position / relocations / operands.
- **Rule condition**: Certain broadness / resilience to changes in malware code, not too rigid.

Finding unique code

- Identify relevant code reuse between lots of binaries.
  - Exclude goodware code.
  - Exclude “forks” of the malware family.

- We will handle this with a code search engine, it allows us to:
  - Create “code-based” signatures first, then transform them into YARA rules.
  - Pre-validation of signatures.
  - Scale to >thousands binaries.

Normalized code

```
55              push ebp
8BEC            mov ebp, esp
B858160000      mov eax, 0x1658
E88F050000      call 0x59c
53              ush ebx
56              push esi
FF7510          push dword ptr [ebp + 0x10]
8D85D8F5FFFF    lea eax, [ebp - 0xa28]
50              push eax
8B4514          mov eax, dword ptr [ebp + 0x14]
E888B2FFFF      call 0xffffb2a9
8D4704          lea eax, [edi + 4]
50              push eax
8D85F0FBFFFF    lea eax, [ebp - 0x410]
50              push eax
B800020000      mov eax, 0x200
E873B2FFFF      call 0xffffb2a9
8D8704060000    lea eax, [edi + 0x604]
50              push eax
8D85E8F9FFFF    lea eax, [ebp - 0x618]
```

→

```
55
8B EC
B8 58 16 00 00
E8 ?? ?? ?? ??
53
56
FF 75 ??
8D 85 ?? ?? ?? ??
50
8B 45 ??
E8 ?? ?? ?? ??
8D 47 ??
50
8D 85 ?? ?? ?? ??
50
B8 00 02 00 00
E8 ?? ?? ?? ??
8D 87 ?? ?? ?? ??
50
```

Normalized code with mkYARA

```python
import codecs
from capstone import CS_ARCH_X86, CS_MODE_32
from mkyara import YaraGenerator

gen = YaraGenerator("normal", CS_ARCH_X86, CS_MODE_32)
gen.add_chunk(codecs.decode("6830800000E896FEFFFFC3", "hex"), offset=0x100)
rule = gen.generate_rule()
rule_str = rule.get_rule_string()
print(rule_str)
```

https://github.com/fox-it/mkYARA

→

```
rule generated_rule
{
	meta:
		generated_by = "mkYARA - By Jelle Vergeer"
		date = "2023-04-07 08:48"
		version = "1.0"

	/*
	0x100 6830800000				push 0x8030
	0x105 E896FEFFFF				call 0xfffffa0
	0x10a C3					ret
	 */
	strings:
		$chunk_1 = {
			68 30 80 00 00
			E8 ?? ?? ?? ??
			C3
		}

	condition:
		any of them

}
```

Rule condition

- We want a certain broadness and resilience to changes in code.
  - This means we need to add more than just a few functions or basic blocks to the rule.
  - … and have a flexible rule condition, say a 20% “threshold” -> **automation.**

- From our experience in studying code reuse at scale over 1000+ malware families: even small overlaps of 10-20% are enough for high quality identification.

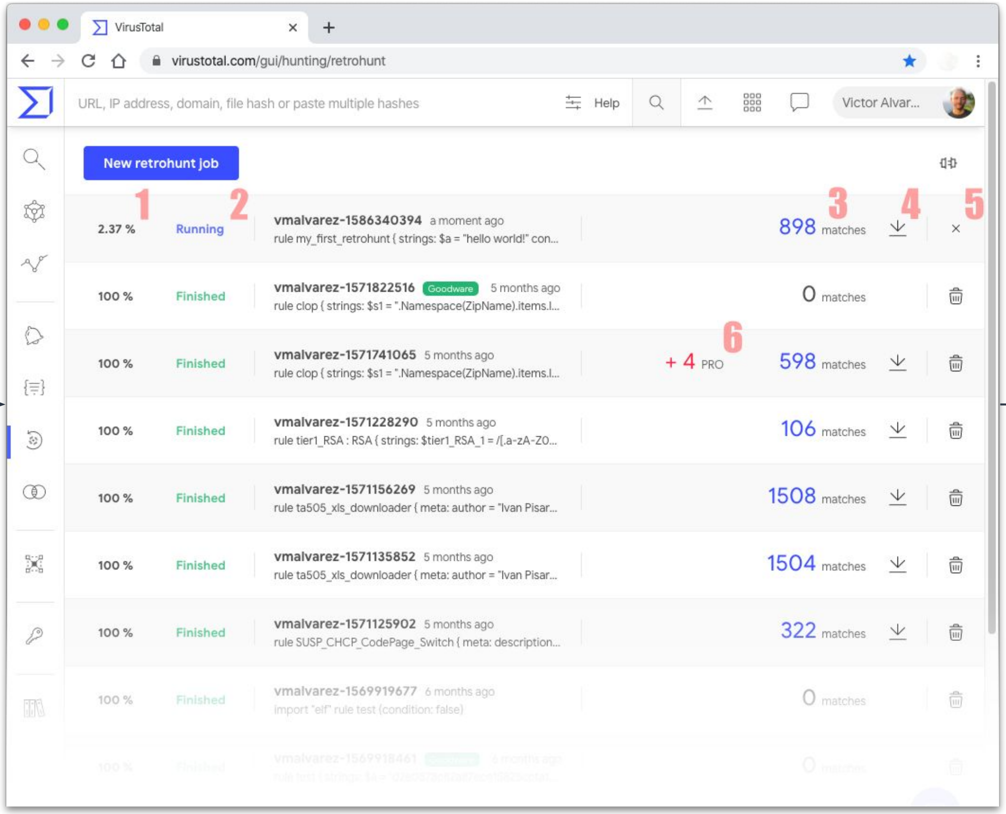
THREATRAY

Code search engine

Finding code reuse at scale

Code search engine - What is it?

Match?

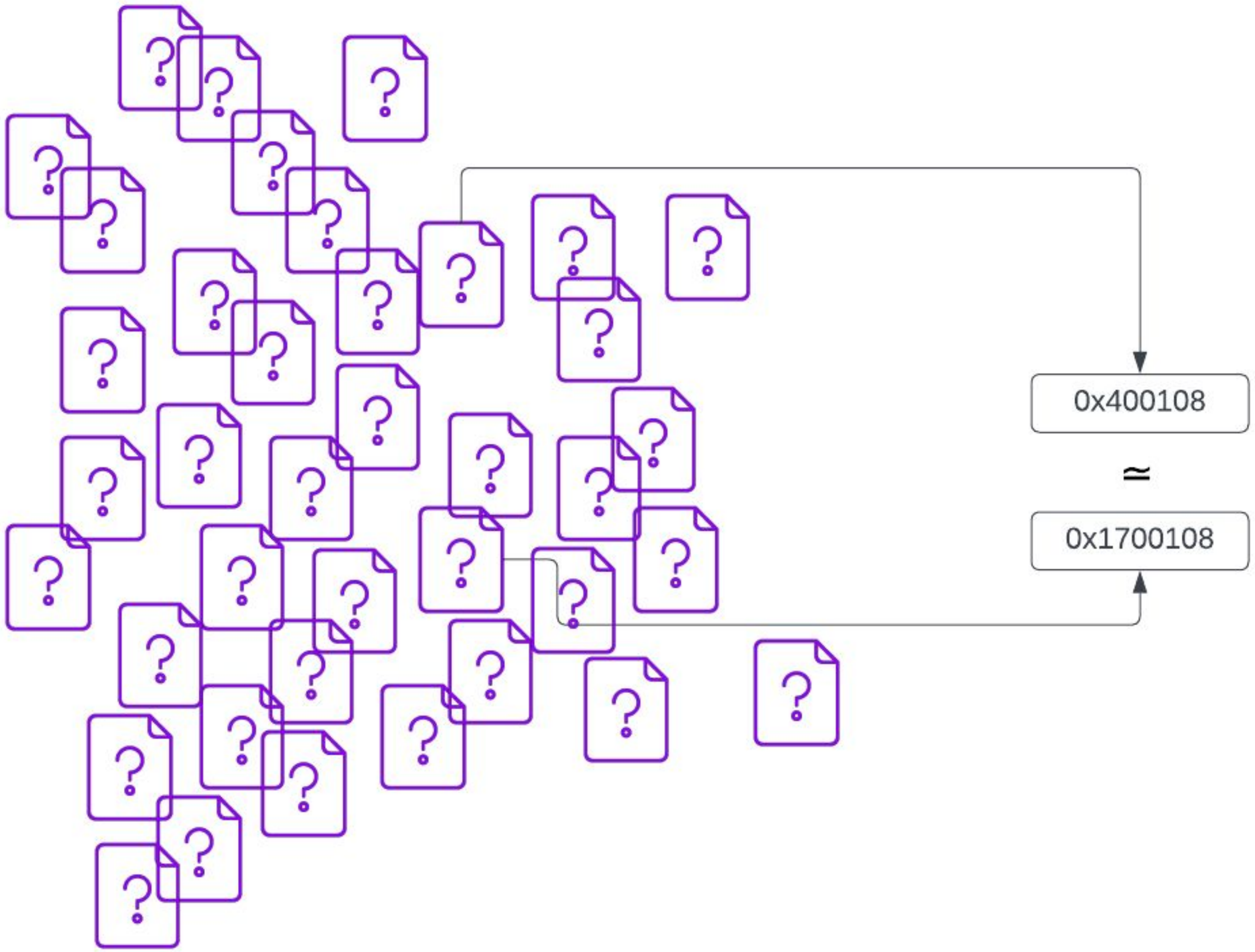


abc.exe	91%
contract.exe	73%

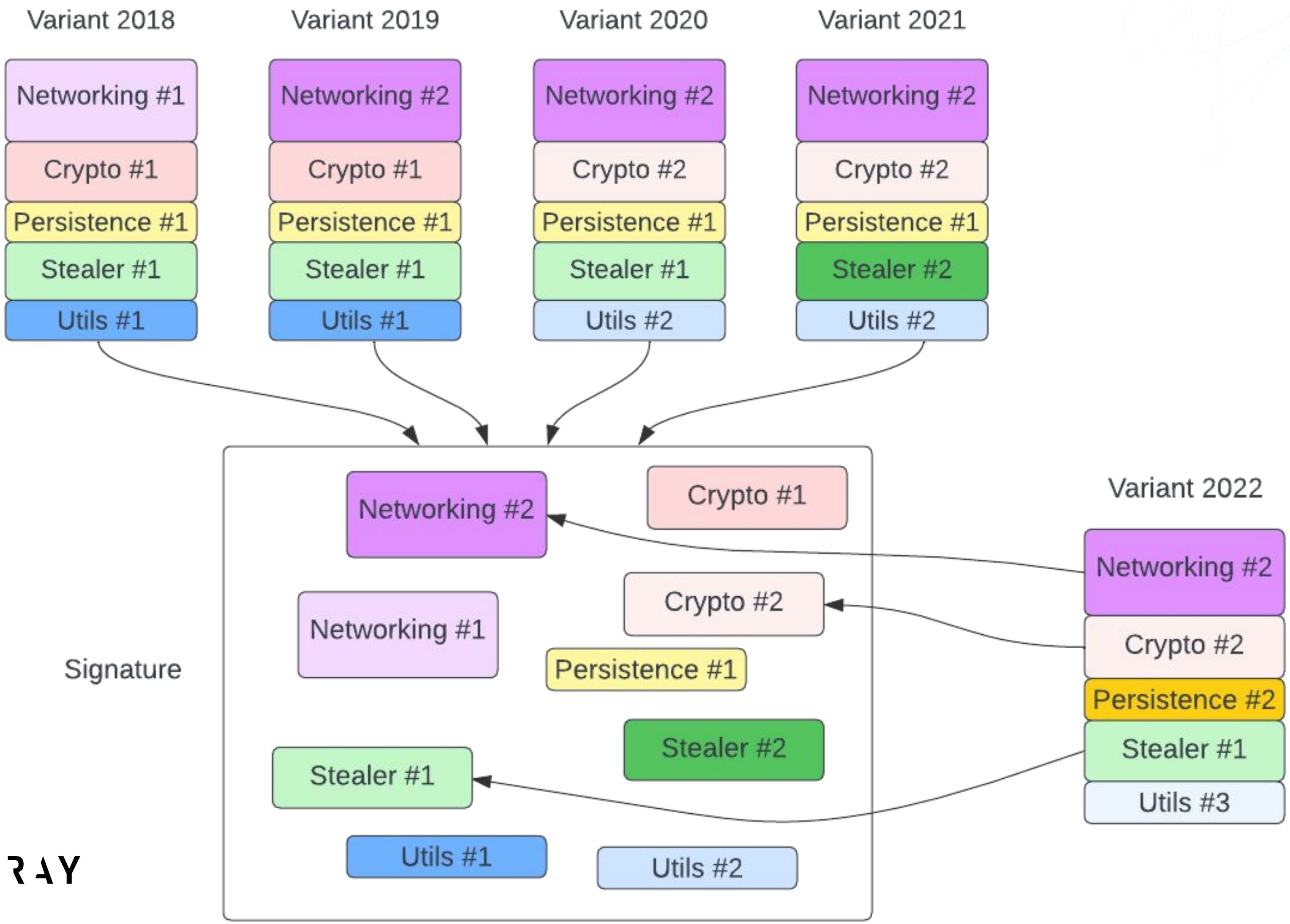
...

THREATRAY

Finding code reuse at scale



Code-based signatures



Threatray Demo

- Code-based signatures
- Native retro-hunting
- Binary OSINT

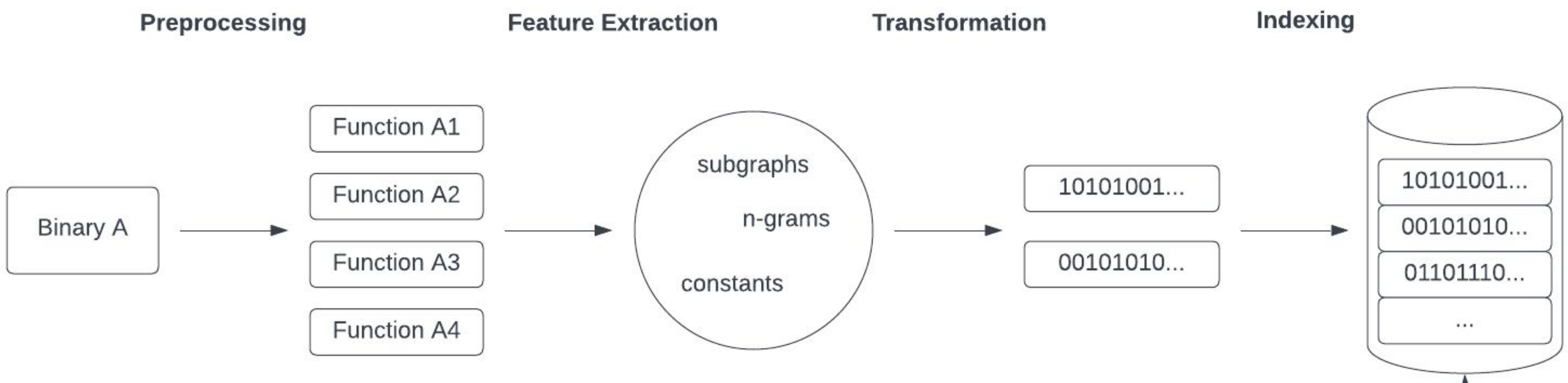
Requirements

- **Granularity**: Need to have a fine granularity of finding code reuse, either function or sub-function level.
- **Accuracy**: Need a high quality code similarity metric to spot code reuse.
- **Scale**: Need to look at dozens to hundreds of binaries of a malware family, at the same time.

Requirements

	Granularity	Accuracy	Scale
ssdeep	❌	❌	✅
Bindiff	✅	✅	❌
Code search engine	✅	✅	✅

Architecture of a code search engine

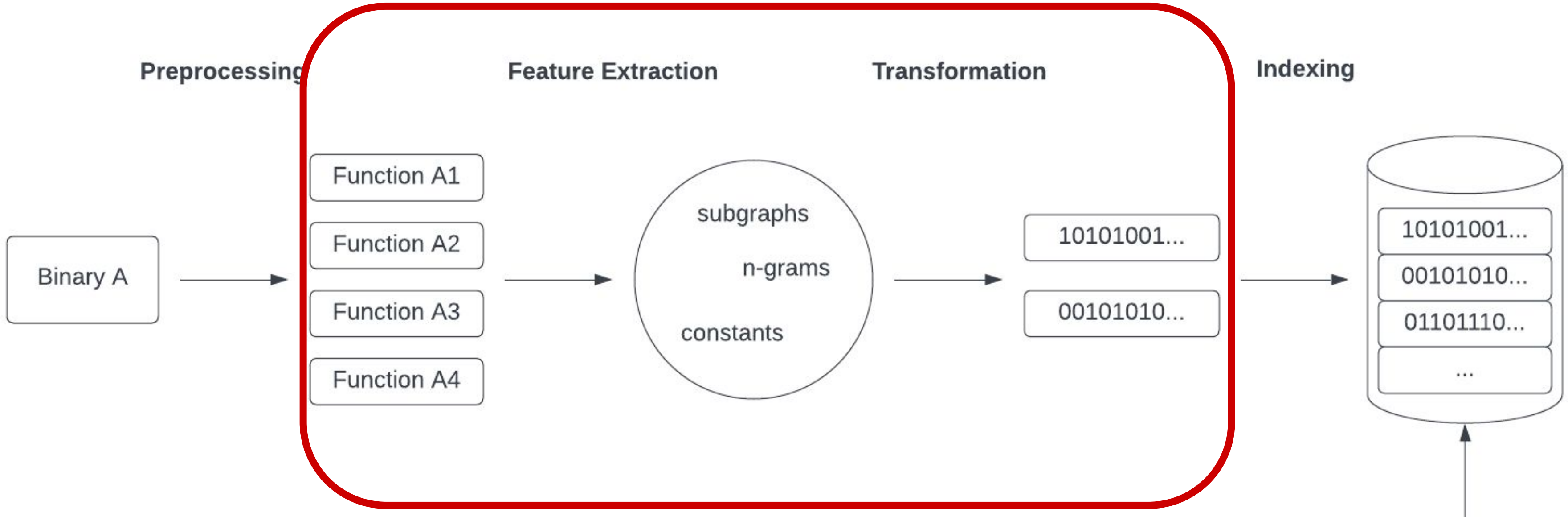


Search

Query: Function A1

Matching Binary	Binary Hash	Matching function	Score
Binary B	169eaff1...	Function B1	1.0
Binary X	934c8fae...	Function X7	0.95
Binary Y	73c59aa0...	Function Y10	0.95
...	...	...	...

Architecture of a code search engine



Search

Query: Function A1

Matching Binary	Binary Hash	Matching function	Score
Binary B	169eaff1...	Function B1	1.0
Binary X	934c8fae...	Function X7	0.95
Binary Y	73c59aa0...	Function Y10	0.95
...	...	...	...

The core parts of a code search engine

- Search granularity
- Code similarity metric
- Distance preserving transformation

Search granularity

- **Binary**
  - Suitable for retro-hunting, binary OSINT.

- **Function**
  - Isolated piece of code with semantic value.
  - Reuse is largely triggered by developers.

- **Basic block**
  - Smallest unit of code with semantic value.
  - Reuse is largely triggered by compilers.

Code similarity metric

```
https://evil.com/api17.php

≃

https://evil.com/api28.php
```

```
b60f ff55 458b 8d08
00e2 b60f 1104 b60f
01ba 0000 d100 0fe2
b60f fe55 e852 fe96

?

b332 ff55 458b 8d08
00e2 b60f f099 b60f
01ba 9999 d100 ab2d
b60f 55fe e852 fe96
```

What is a good code similarity metric?

- **Semantic** vs. **syntactic** code similarity.

- Resilient against differences in code generation:
  - Position dependence
  - Compiler versions
  - Compiler optimization levels
  - Word size (32/64-bit)
  - CPU architectures

- Resilience against minor differences in source code (=approximate semantic similarity).

Instruction sequence similarity

- Basic block -> normalized code
  - BinLex calls them “traits”.

```
55                  push ebp
8BEC                mov ebp, esp
B858160000          mov eax, 0x1658
E88F050000          call 0x59c
53                  ush ebx
56                  push esi
FF7510              push dword ptr [ebp + 0x10]
8D85D8F5FFFF        lea eax, [ebp - 0xa28]
50                  push eax
8B4514              mov eax, dword ptr [ebp +
0x14]
E888B2FFFF          call 0xffffb2a9
8D4704              lea eax, [edi + 4]
50                  push eax
8D85F0FBFFFF        lea eax, [ebp - 0x410]
50                  push eax
B800020000          mov eax, 0x200
E873B2FFFF          call 0xffffb2a9
8D8704060000        lea eax, [edi + 0x604]
50                  push eax
8D85E8F9FFFF        lea eax, [ebp - 0x618]
```

```
55
8B EC
B8 58 16 00 00
E8 ?? ?? ?? ??
53
56
FF 75 ??
8D 85 ?? ?? ?? ??
50
8B 45 ??
E8 ?? ?? ?? ??
8D 47 ??
50
8D 85 ?? ?? ?? ??
50
B8 00 02 00 00
E8 ?? ?? ?? ??
8D 87 ?? ?? ?? ??
50
```

- Pros
  - Simple and explainable approach.
  - Fine-grained similarity metric.

- Cons
  - Weak towards even minor changes in code generation.
  - Requires lots of data per function, which makes it harder to scale.

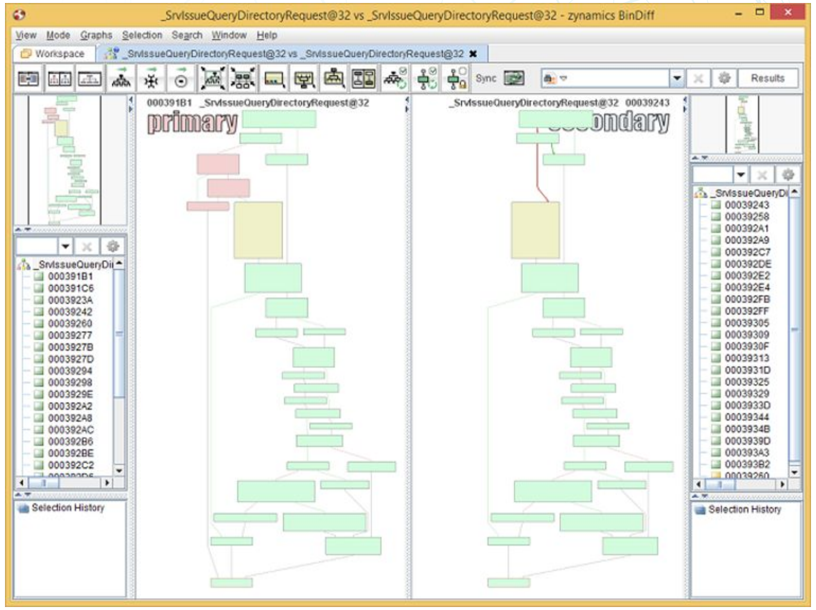
Control flow graph similarity

- Related to the graph isomorphism problem, which is hard to solve.

- Similarity is matching basic blocks from graph A to graph B.

- Pros
    - Good resilience towards code generation changes.
    - Can be made to scale to 100M+ functions.

- Cons
    - Quality depends a lot on feature extraction process.
    - Usually no “sub-function” similarity.



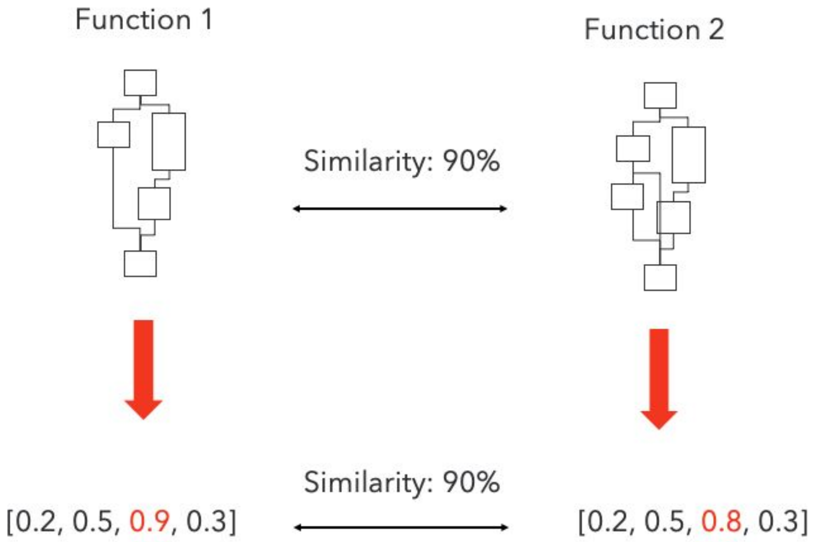
Distance preserving transformation

- We need to be able to compare code efficiently at scale.
  - Instruction sequence similarity -> explosion of storage.
  - Control flow graph similarity -> CPU-intensive.

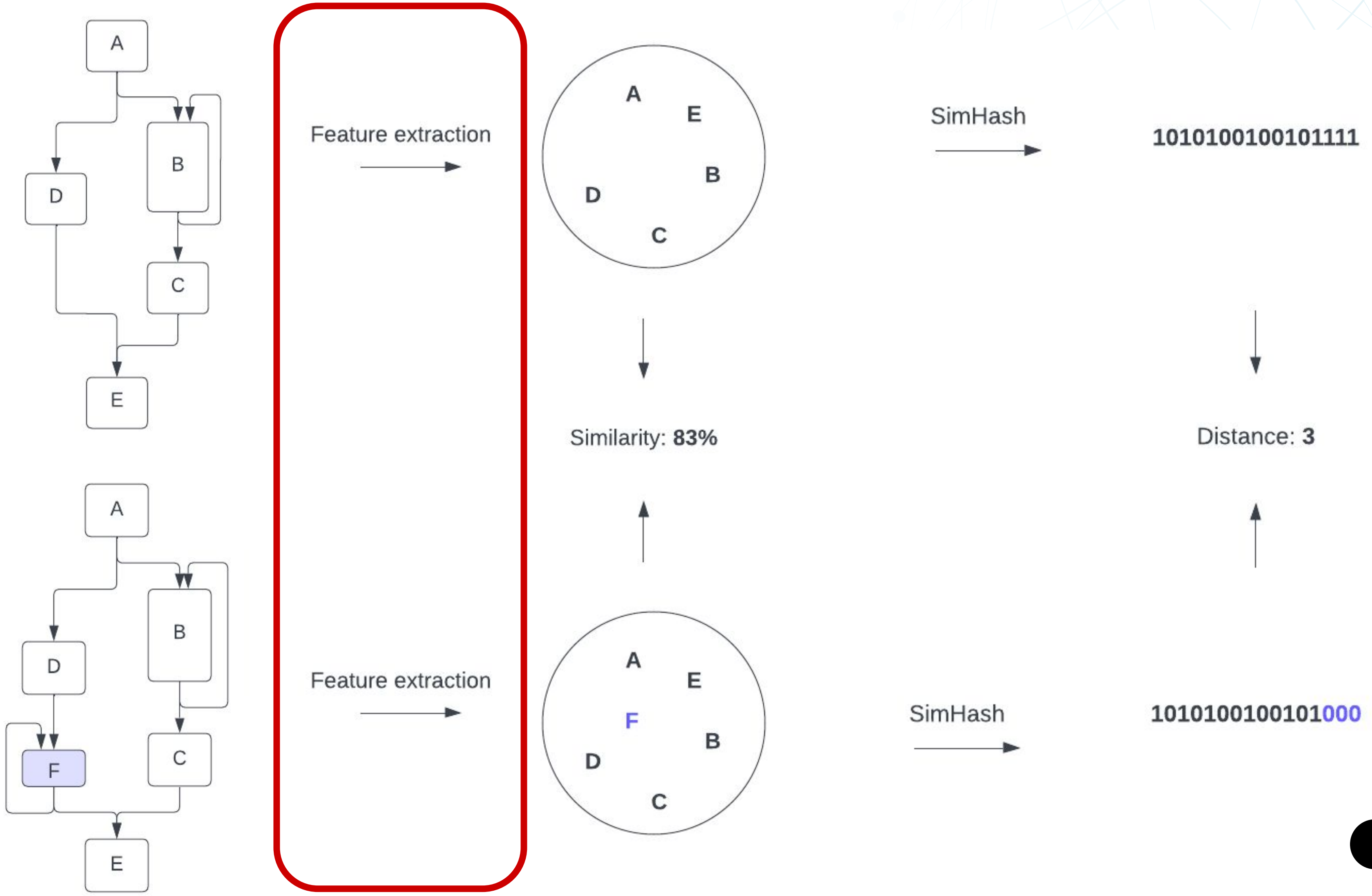
- The solution is to have a **distance preserving transformation.**

- The goal is to be compute the similarity of the representation after transformation, with the properties of:
  - Similarity is easy to compute.
  - Similarity in representation = similarity in actual code.

- This is usually **locality-sensitive hashing** (fuzzy hashing) or **embedding** into low-dimensional vector spaces.



SimHash for functions

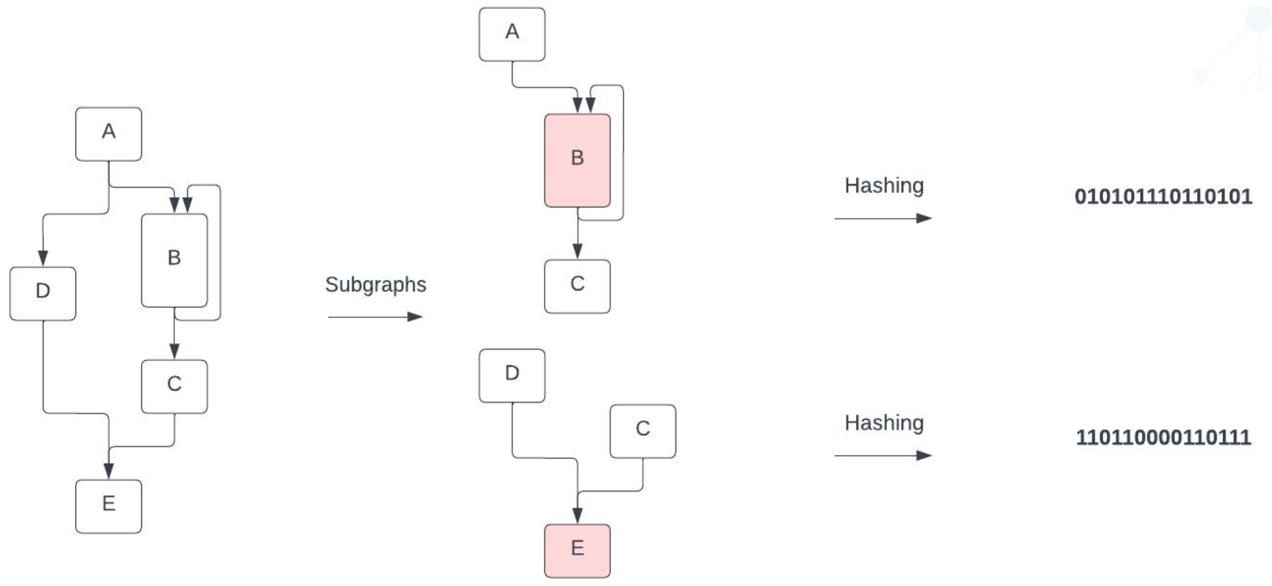


From control flow graph features to SimHash

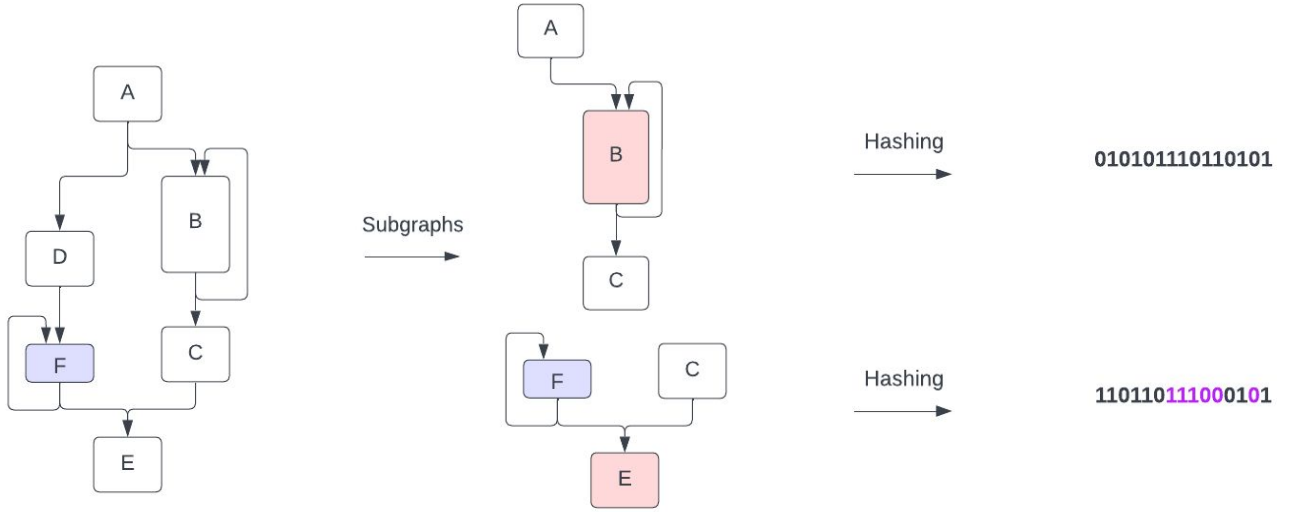
- We need to extract features that adhere to the requirements outlined before, e.g. resilient against changes in code generation.

- FunctionSimSearch uses three types of features that achieve this:
  - **Subgraphs** of the control flow graph.
  - **N-grams** from mnemonics of the instruction sequence.
  - **Constants** from the function.

Subgraphs



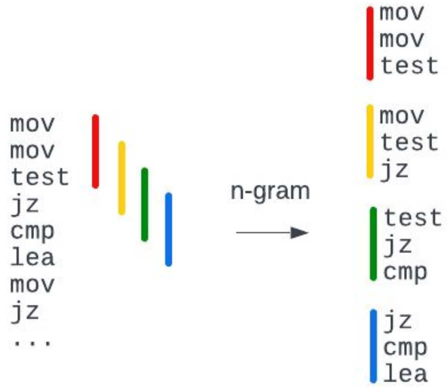
A
B
D
C
E
Subgraphs
A
B
C
D
C
E
Hashing
Hashing
010101110110101
110110000110111



A
B
D
F
C
E
Subgraphs
A
B
C
F
C
E
Hashing
Hashing
010101110110101
110110111000101

n-grams of mnemonics

```
.text:00007FF790D424D0                          ; sub_7FF790D45160+13D1↓p
.text:00007FF790D424D0             mov     r9, rdx
.text:00007FF790D424D3             mov     r10, rcx
.text:00007FF790D424D6             test    rdx, rdx
.text:00007FF790D424D9             jz      short loc_7FF790D42526
.text:00007FF790D424DB             cmp     word ptr [rdx+2], 0
.text:00007FF790D424E0             lea     rax, [rdx+2]
.text:00007FF790D424E4             mov     r8d, 1
.text:00007FF790D424EA             jz      short loc_7FF790D424FD
.text:00007FF790D424EC             nop     dword ptr [rax+00h]
.text:00007FF790D424F0
.text:00007FF790D424F0 loc_7FF790D424F0:                ; CODE XREF: sub_7FF790D424D0+2B↓j
.text:00007FF790D424F0             inc     r8d
.text:00007FF790D424F3             lea     rax, [rax+2]
.text:00007FF790D424F7             cmp     word ptr [rax], 0
.text:00007FF790D424FB             jnz     short loc_7FF790D424F0
.text:00007FF790D424FD
.text:00007FF790D424FD loc_7FF790D424FD:                ; CODE XREF: sub_7FF790D424D0+1A↑j
.text:00007FF790D424FD             test    r8d, r8d
.text:00007FF790D42500             jz      short loc_7FF790D42526
.text:00007FF790D42502             sub     r9, rcx
.text:00007FF790D42505             mov     edx, r8d
.text:00007FF790D42508             nop     dword ptr [rax+rax+00000000h]
.text:00007FF790D42510
.text:00007FF790D42510 loc_7FF790D42510:                ; CODE XREF: sub_7FF790D424D0+50↓j
.text:00007FF790D42510             movzx   eax, word ptr [r9+rcx]
.text:00007FF790D42515             mov     [rcx], ax
.text:00007FF790D42518             lea     rcx, [rcx+2]
.text:00007FF790D4251C             sub     rdx, 1
.text:00007FF790D42520             jnz     short loc_7FF790D42510
.text:00007FF790D42522             mov     rax, r10
.text:00007FF790D42525             retn
```



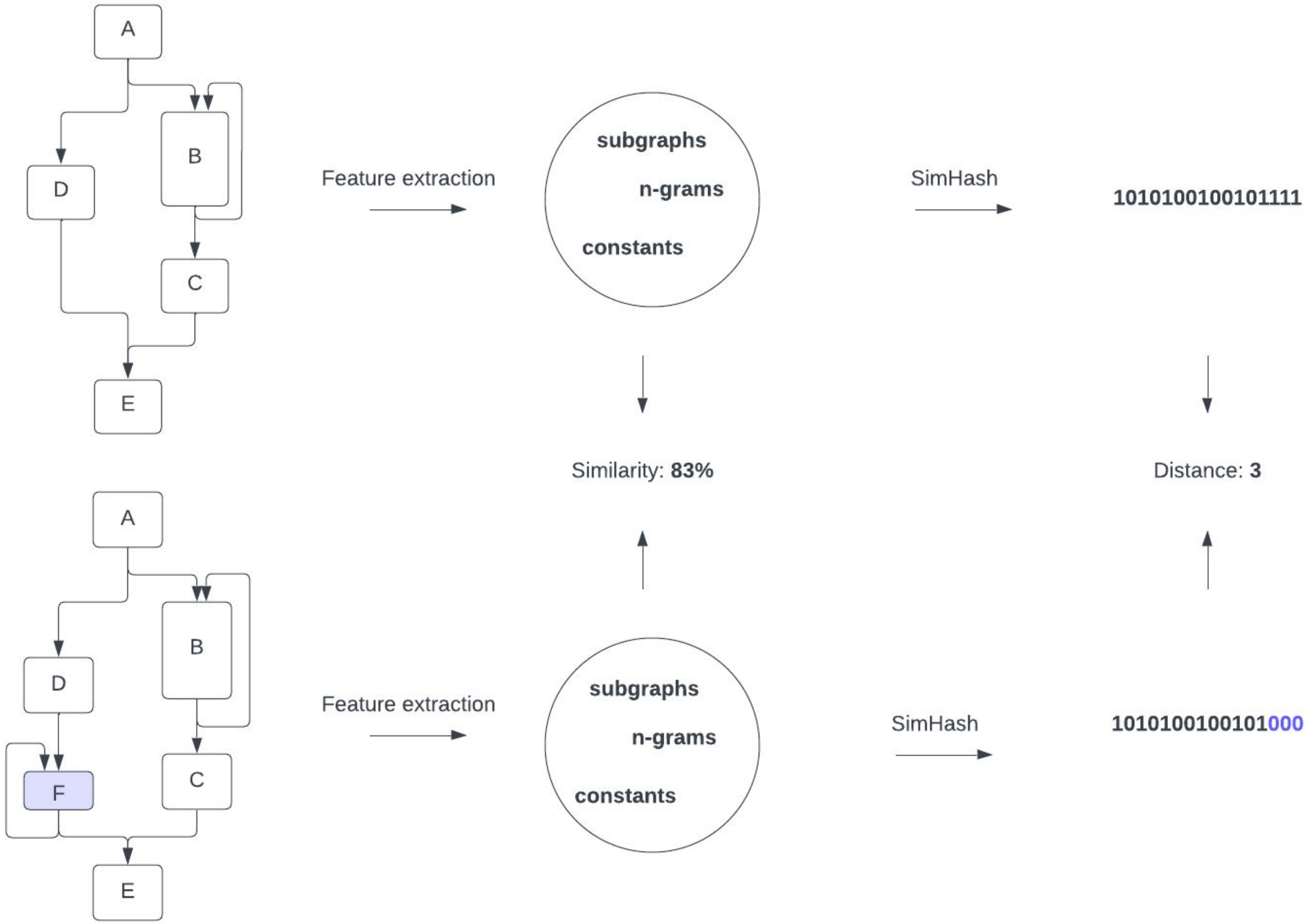
Constants from the instruction sequence

- Certain constants are often unchanged by the compiler and thus are relevant identifiers for code reuse.

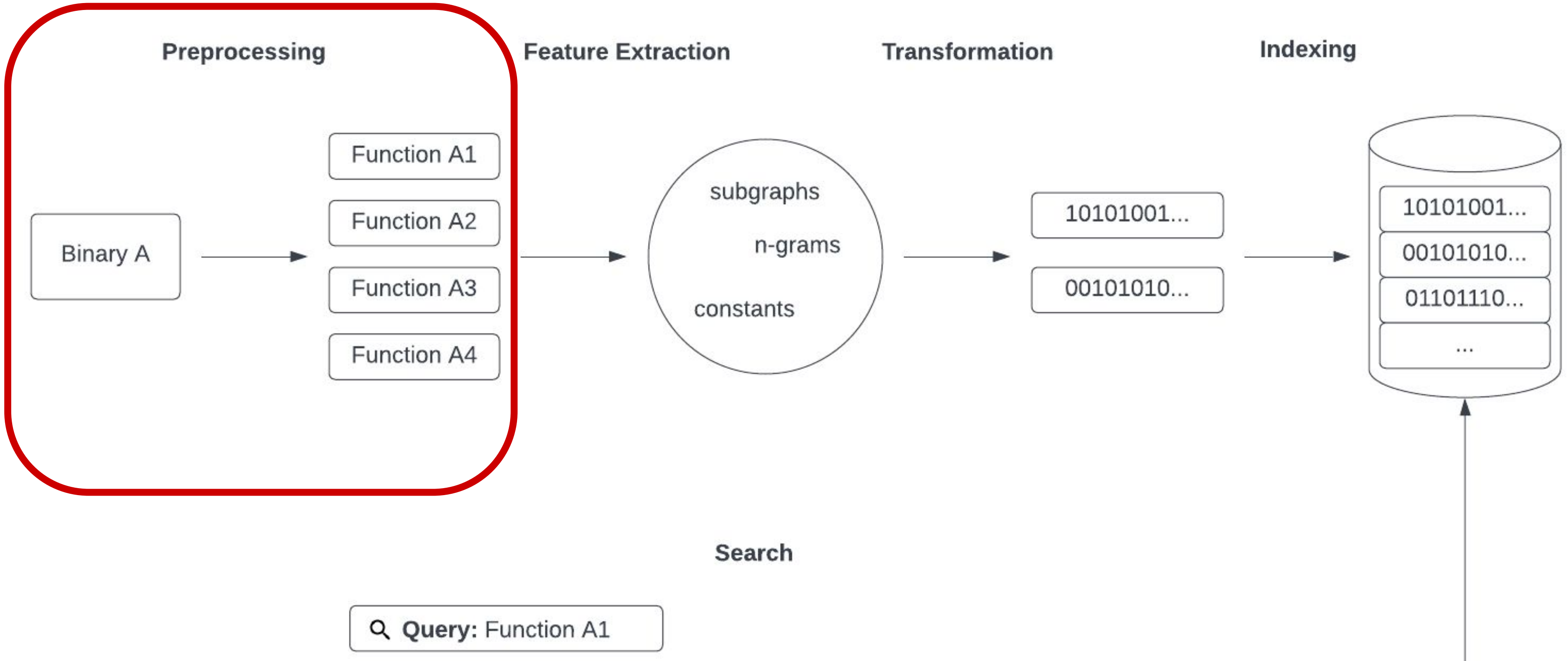
- FunctionSimSearch only considers constants that are:
  - greater than 0x4000
    OR
  - divisible by 4 and greater than 10

- With the idea of removing stack offsets, which aren't good features.

SimHash for functions

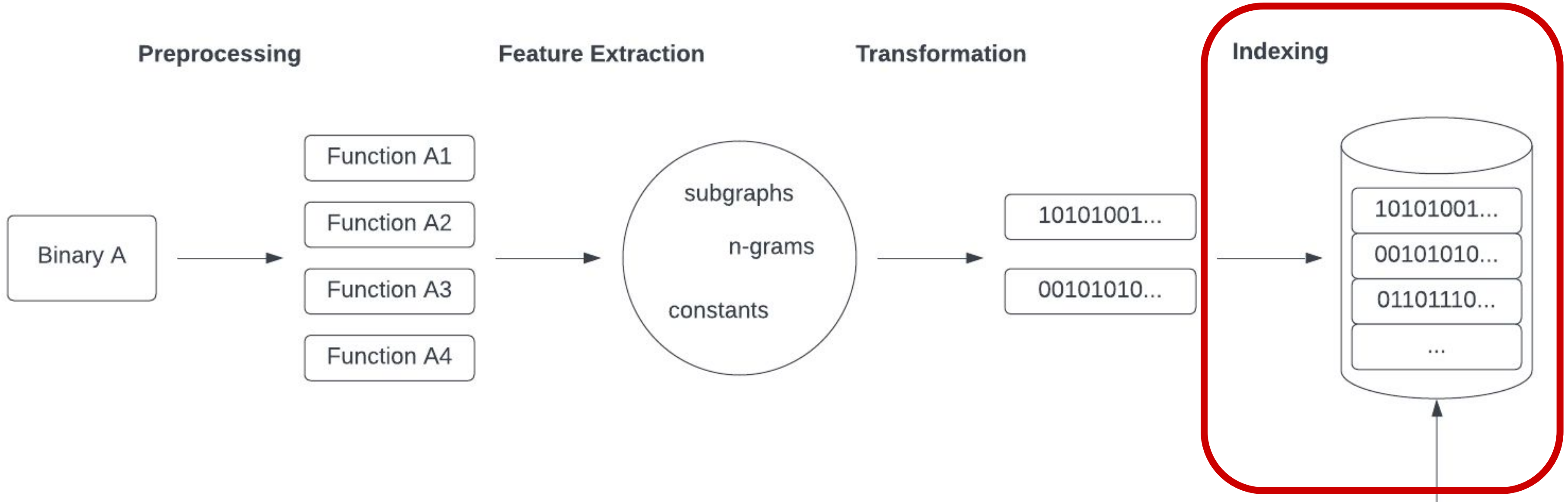


Building a code search engine



Matching Binary	Binary Hash	Matching function	Score
Binary B	169eaff1...	Function B1	1.0
Binary X	934c8fae...	Function X7	0.95
Binary Y	73c59aa0...	Function Y10	0.95
...	...	...	...

Building a code search engine

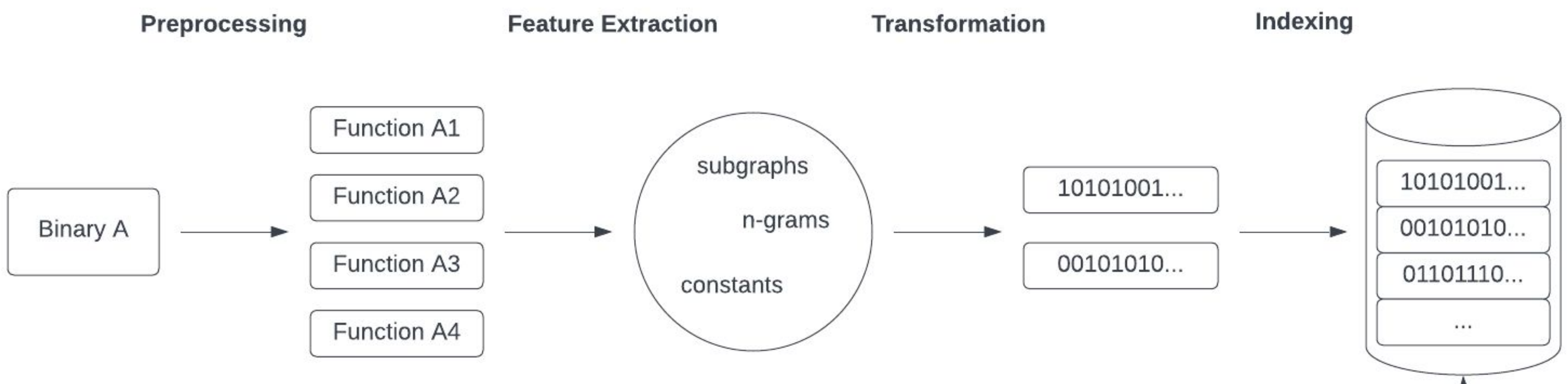


Search

Query: Function A1

Matching Binary	Binary Hash	Matching function	Score
Binary B	169eaff1...	Function B1	1.0
Binary X	934c8fae...	Function X7	0.95
Binary Y	73c59aa0...	Function Y10	0.95
...	...	...	...

Building a code search engine



Search

Query: Function A1

Matching Binary	Binary Hash	Matching function	Score
Binary B	169eaff1...	Function B1	1.0
Binary X	934c8fae...	Function X7	0.95
Binary Y	73c59aa0...	Function Y10	0.95
...	...	...	...

Example 1 - BinLex

- **Preprocessing**: Disassemble with capstone.
- **Feature extraction**: Extract normalized instruction sequences (=“traits”).
- **Transformation:** Apply cryptographic hash to traits.
- **Indexing**: Lookup that maps trait hashes to traits, no similarity metric.
- **Search**: Index lookup on input traits IDs (=hash), return metadata.

Example 2 - FunctionSimSearch

- **Preprocessing**: Disassemble with dyninst.
- **Feature extraction**: Extract subgraphs, n-grams and constants from control flow graphs of functions.
- **Transformation:** Apply SimHash to extracted features.
- **Indexing**: Partition SimHashes into buckets of inverted indices.
- **Search**: Input function (=SimHash) is matched through the index and returns matching functions and metadata.

THREATRAY

Building a YARA rule creation pipeline

YARA rule creation pipeline

BinLex

- **Index a set of binaries**
  - 1) Parse binary
  - 2) Extract traits (**Normalized**)
  - 3) Filter traits (e.g by number of instructions)
  - 4) Index filtered traits
- **Create Code Signature**
  - 5) Add samples shared traits to the Code signature
  - 6) Remove whitelisted traits from the Code Signature
  - 7) Remove shared traits from another Code Signatures
  - 8) Test Code Signature
- **Yara rule creation**
  - 9) Generate Yara rule with Code Signature traits
  - 10) Test Yara rule (adjust condition)

FunctionSimSearch

- **Index a set of binaries**
  - 1) Parse binary
  - 2) Extract functions (**Not normalized**)
  - 3) Index functions using SimHash
- **Create Code Signature**
  - 4) Select function candidates by similarity**
  - 5) Test wildcarded functions
- **Yara rule creation**
  - 6) Generate Yara rule with wildcarded functions
  - 7) Test Yara rule

** if the shared funcs aren't 100% similar, BinLex will be used for creating the wildcarded function with traits.