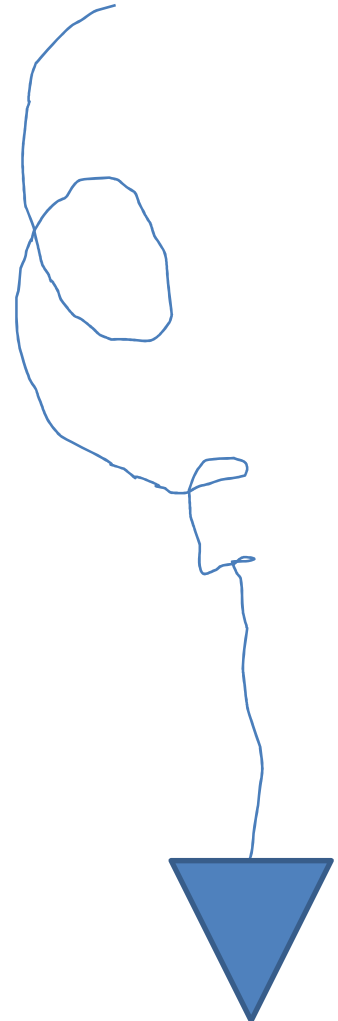
Ut cognitione visus: ut ipso intellecto

BinNavi v2

Tiago Assumpção, zynamics

Agenda

- Speech
- The *prolix* and the synthetic
- Artificial languages
- Programming languages
- Machine code
- Reverse engineering
- Program analysis
- Real life & BinNavi
- Demo
- APPENDIX

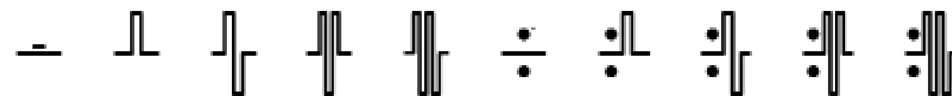


Speech: structure

A - Z :

a - z :

0 - 9 :



Speaking: denotation / conotation

- *Default interpretation of the speaker's utterance is normally understood to mean salient meaning intended by the speaker, or presumed by the addressee to have been intended, and recovered (a) without the help of inference from the speaker's intentions or (b) without conscious inferential process altogether*

***Defaults in Semantics and Pragmatics*, Stanford Encyclopedia of Philosophy**

The *prolix* and the synthetic

- *Aufsichtsratsmitgliederversammlung*

- "On-view-council-with-limbs-gathering" meaning "meeting of members of the supervisory board" ("with" and "limb" forming a derivation that is the German word for "member")

The *prolix* and the synthetic

- *Käyttäytyessään tottelemattomasti oppilas saa jälki-istuntoa*

- "Should he/she behave in an insubordinate manner, the student will get detention." Structurally: behaviour(present/future tense)(of his/hers) obey(without)(in the manner/style) studying(he/she who (should be)) gets detention(some).

Artificial languages

- Whenever something special is planned to mean...
  - An utterance structure, intention, foist self-stem

Artificial languages: examples

- Symbolic logic
- “*One who hopes*”
- Your programming language of choice

Programming languages

- Programming languages should:
  - Describe anything an abstract machine could possibly compute
  - Be easy to manipulate and to understand to the human being

Programming languages

- For instance;

```
static unsigned int           c = 0;

void
f (void)
{
              unsigned int    a = 12345;
              unsigned int    b = 67890;


              c = a + b;
}

int
main (void)
{
              f ();

              return          c;
}
```

Programming languages

- From description to execution, one program must be transformed from a high-level, human-driven, into semantically equivalent machine speech

Machine code

- Machine code may be regarded as a primitive (and cumbersome) programming language or as the lowest-level representation of a compiled and/or assembled computer program

What Wikipedia tells us about **Machine Code**

Machine code

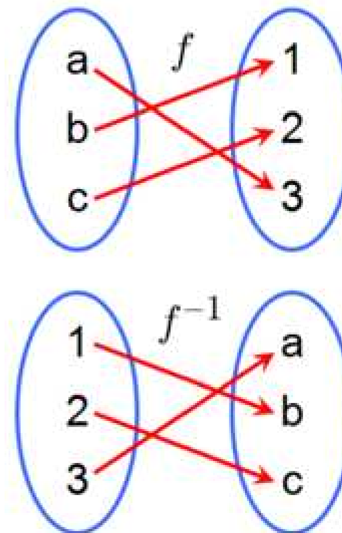
- To the machine, size is everything
- The same as human friendly, but, way more compact or synthetic

Machine code

```
08048350 <f>:
 8048350:       55
 8048351:       89 e5
 8048353:       5d
 8048354:       c7 05 60 95 04 08 6b
 804835b:       39 01 00
 804835e:       c3
 804835f:       90

 08048360 <main>:
 8048360:       8d 4c 24 04
 8048364:       83 e4 f0
 8048367:       ff 71 fc
 804836a:       b8 6b 39 01 00
 804836f:       55
 8048370:       89 e5
 8048372:       51
 8048373:       c7 05 60 95 04 08 6b
 804837a:       39 01 00
 804837d:       59
 804837e:       5d
 804837f:       8d 61 fc
 8048382:       c3
```

Reverse engineering



Reverse engineering: the problem

- An enormous amount of information described by means of an utterly condensed structure
- Information collection, organization and comprehension represent an epic movement to the [human] interpreter.
- Often an epic fail.

Program analysis

- Program analysis allows for satisfactory automated comprehension of a given program
- Usually followed by transformations that may extract certain *things* and create different representations by means of *well-formed formulæ*
- We're concerned with:
  - Structure
  - Meaning
  - Objects
  - Types
  - Relationships

Program analysis: example

- Compilation, optimization, obfuscation, *others*
  - A set of program analysis & transformations throughout different phases, each with specific aims

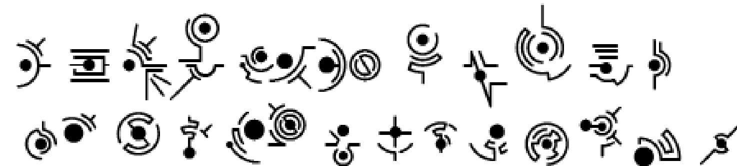
Program analysis: automated RE

- Computer [imperative]: assuming a domain of discourse, read and interpret a stream *S*, and bring it back into its *original* representation or to an intermediate form of choice

Real life & BinNavi

- Real life ain’t never so sweet and *formalisms* alone are just not enough: have the right ideas assisted by the proper techniques

A - Z :



a - z :

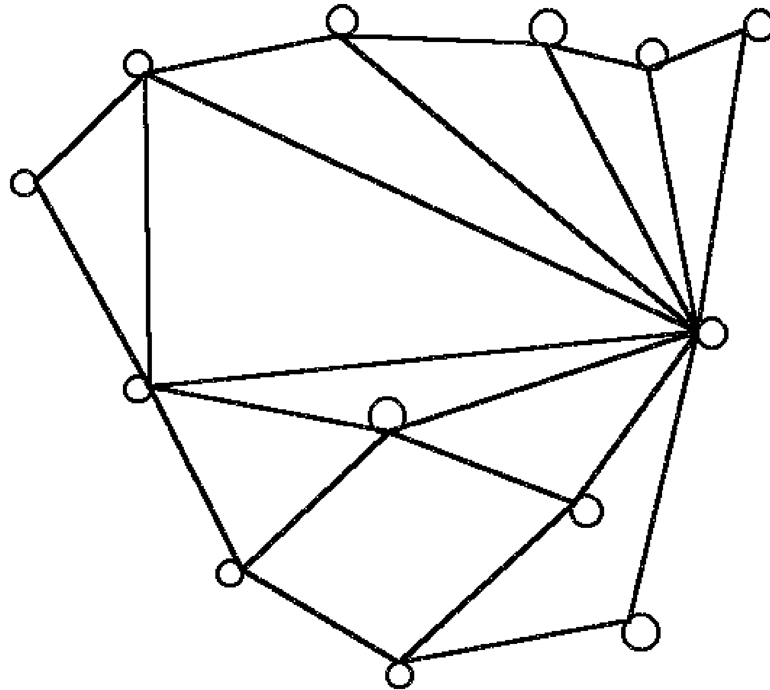
0 - 9 :

Real life & BinNavi: theory

- After Patrick Cousot;
- Sound aproximation of the semantics of algorithm descriptions derived from monotonic functions over partially ordered sets - especially lattices (or, the algebraic equivalent, galois connections)
- Often used to describe the partial execution of a computer program and its semantics

Real life & BinNavi: visualization

- Which can be presented as directed graphs



Real life & BinNavi

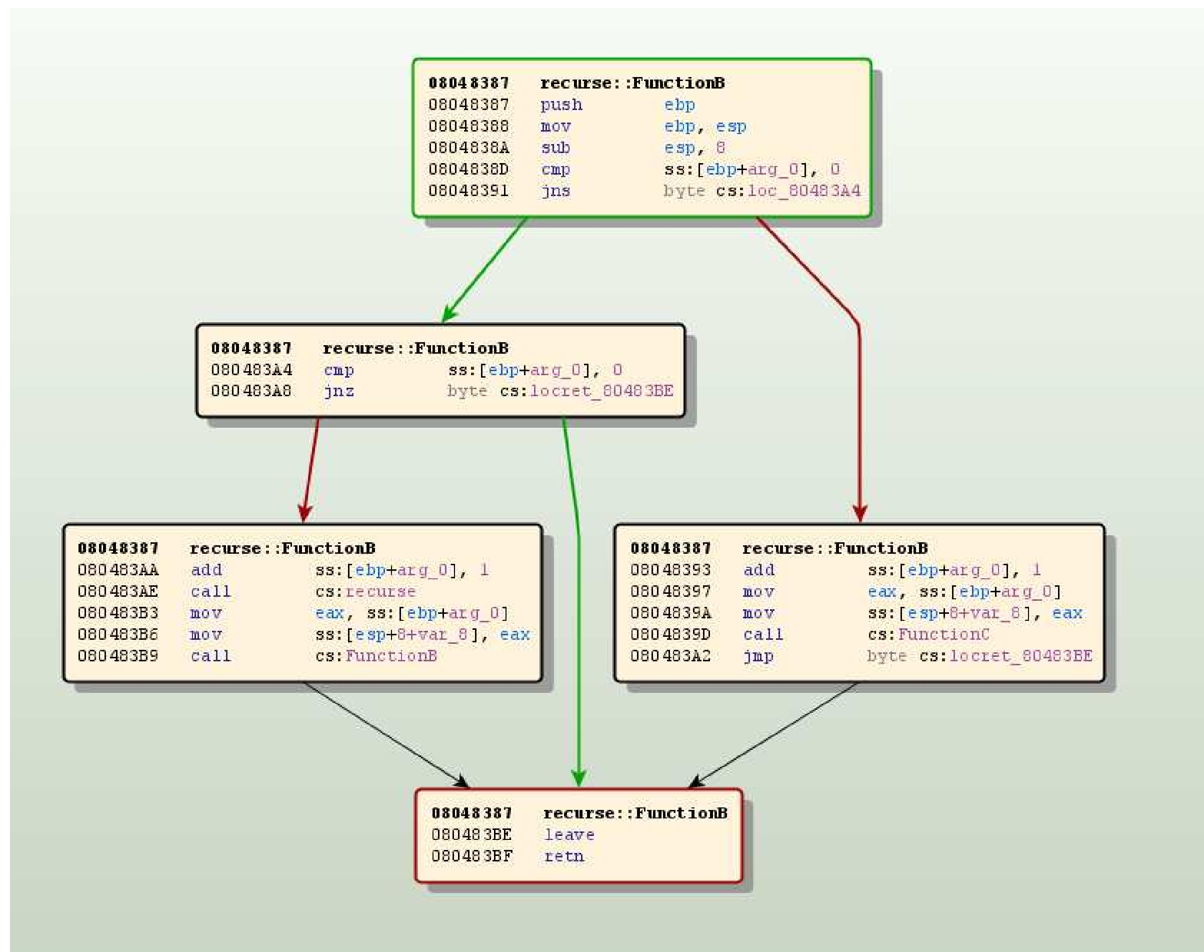
- Pierce believed logic was a branch of semiotics
- I am certain that visualization is the culminating of logic

Ut cognitione visus: ut ipso intellecto

Real life & BinNavi: before

```
.text:08048387 FunctionB      proc near            ; CODE XREF: FunctionC+19p
.text:08048387                                     ; FunctionB+32p ...
.text:08048387
.text:08048387 var_8          = dword ptr -8
.text:08048387 arg_0          = dword ptr  8
.text:08048387
.text:08048387                push    ebp
.text:08048388                mov     ebp, esp
.text:0804838A                sub     esp, 8
.text:0804838D                cmp     [ebp+arg_0], 0
.text:08048391                jns     short loc_80483A4
.text:08048393                add     [ebp+arg_0], 1
.text:08048397                mov     eax, [ebp+arg_0]
.text:0804839A                mov     [esp+8+var_8], eax
.text:0804839D                call    FunctionC
.text:080483A2                jmp     short locret_80483BE
.text:080483A4 ; ---------------------------------------------------------------------------
.text:080483A4
.text:080483A4 loc_80483A4:                        ; CODE XREF: FunctionB+Aj
.text:080483A4                cmp     [ebp+arg_0], 0
.text:080483A8                jnz     short locret_80483BE
.text:080483AA                add     [ebp+arg_0], 1
.text:080483AE                call    recurse
.text:080483B3                mov     eax, [ebp+arg_0]
.text:080483B6                mov     [esp+8+var_8], eax
.text:080483B9                call    FunctionB
.text:080483BE
.text:080483BE locret_80483BE:                     ; CODE XREF: FunctionB+1Bj
.text:080483BE                                     ; FunctionB+21j
.text:080483BE                leave
.text:080483BF                retn
.text:080483BF FunctionB      endp
```

Real life & BinNavi: after



Real life & BinNavi

- From the simplicity of drawing a small control flowgraph into the sophistication of the proper arragements:

"It's all about finding the right lattices"

– Halvar Flake

Demo: see it thyself

1. Visual navigation
2. Information filtering: from massive modules and immense callgraphs into *selected* flowgraphs
3. Pathfinding & Dominance
4. Framework / API / scripting
5. REIL & register tracking
6. MonoREIL & lattices
7. Finding, understanding and exploiting a real bug

APPENDIX: input tracking

1. A flawed procedure is known to exist at point *p* in program *P*. Verify that the involved variables are controlled by user's input at a set of possible known points

2. A flawed procedure is known to exist at point *p* in program *P*. It was verified that the involved variables are controlled by user's input due to *input crafting* (e.g.: fuzz-testing results). From a known entry point, determine the code path and the related data flow that triggered the bug

Input tracking: case 1

Type: static analysis

Premises:

- a flawed procedure exists at a point p in program P;
- a set of variables v, flaw inductive towards p, is known (usually operands at p);
- a set of entry points α is known

Input tracking: case 1

Determine:

1. there exists at least one *e* that dominates *p*; such that is in *α*;
2. *β*, the set of all possible paths between *e* and *p*;
3. *γ*, the set of all possible predicates for *v* in the domain *β*

Input tracking: case 1

Solution:

- BinNavi's *Lengauer-Tarjan* implementation to determining basic blocks dominance and dominance frontiers;
- BinNavi's *Path-finding* implementation;
- Psi-SSA [1] - an extension of classic SSA that allows for predicate-aware sparse data flow analysis.

Input tracking: case 1

Solution:

1. $i \leftarrow 0$
2. **for** each q in β
3. $i \leftarrow i + 1$
4. **ψ convert** $q \Rightarrow \gamma[i]$
5. eliminate non-predicates of $\gamma[i]$ from ψ-functions/use-def chain

Input tracking: case 1

Improvements:

1. The application of a similar technique implemented by means of SSI instead of SSA would result in a framework that allows for both forward and backwards dataflow analysis, allowing tracks to be made from point A into B or its inversed order, B into A - quite an useful ability when performing vulnerability analysis.

Input tracking: case 1

Improvements:

2. However the implementation of SSA or SSI variants over BinNavi's REIL would be quite straightforward, a realistic solution would require further adjustments such as adding flow and type alias analysis to the presented solution

Input tracking: case 2

Type: dynamic analysis (tracing) used to reduce state-space; followed by static, intra-procedural, analysis

Premisses:

- *smc_decode_frame* is at 689B5660. This function has some internal logic that iterates through a set of *chunks* to be parsed;
- the vulnerability triggering instruction is at 689B60D7, inside the *triggering_basic_block* (689B60D4);
- there exists at least one *x*, such that *x* is in *chunks*, that satisfies: *HALT |= smc_decode_frame(x)*

Input tracking: case 2

Determine:

1. x;
2. the code path for x;
3. the predicates for x

Input tracking: case 2

Solution:

- BinNavi's *Lengauer-Tarjan* implementation to determining basic blocks dominance and dominance frontiers;
- Psi-SSA [1] - an extension of classic SSA that allows for predicate-aware sparse data flow analysis.

Input tracking: case 2

Solution:

1. **repeat**
2. **for** each x in *chunks*
3. *smc_decode_frame* : $x \rightarrow$ *triggering_basic_block*
4. **until** *HALT*
5. **ψ convert** *smc_decode_frame* → *triggering_basic_block*
6. eliminate non-predicates of *triggering_basic_block* (EAX,EDX) from ψ-functions/use-def chain

Input tracking: enquiries

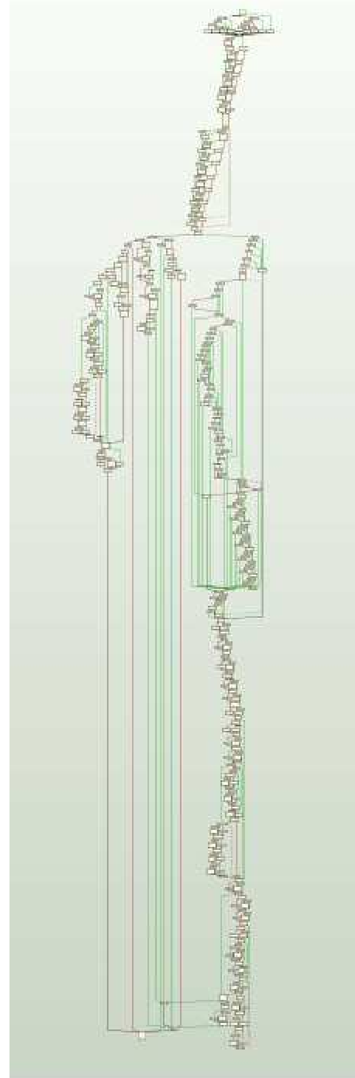
Q: The trace results in deterministic control flow; thus, in this case, the use of Psi-SSA being strictly to determine data flow predication. Why, then, bother about the above when a complete solution for tracking data references throughout code execution could be given by means of tracking, bounded by *type*, memory/register usage (a.k.a. *watch*) from node *A* into node *B*?

Input tracking: enquiries

Example:

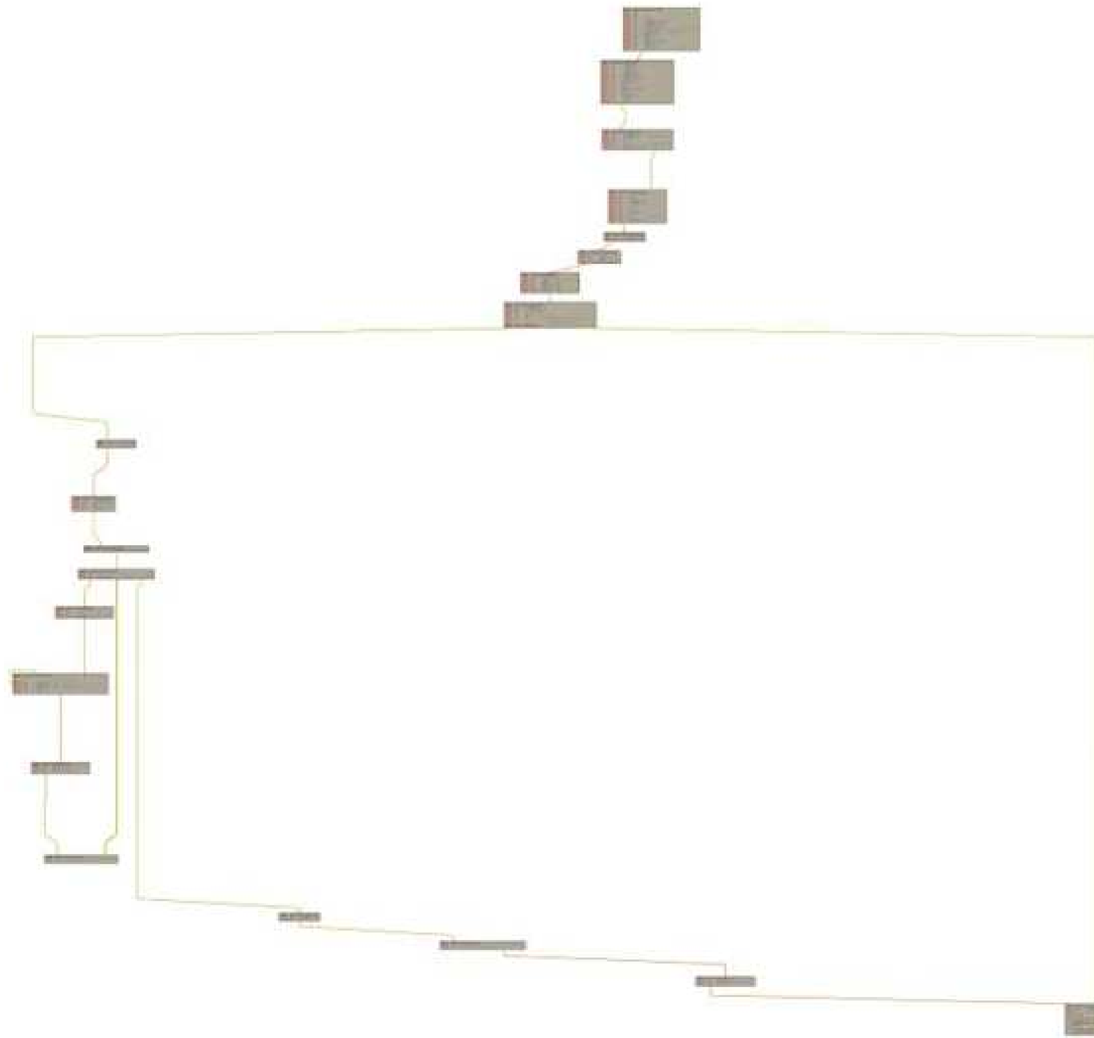
let the objects of arbitrary types { *a, b, c, d, e* } be the input subset for function *F* (*q*{...}, *w*{ *a, x, y, b, c*}, *z, d, e*). Define the subgraph *H* of *F* containing the set of instructions that influence { *a, b, c, d, e* }

Input tracking: enquiries



I. Graph G for function F

Input tracking: enquiries

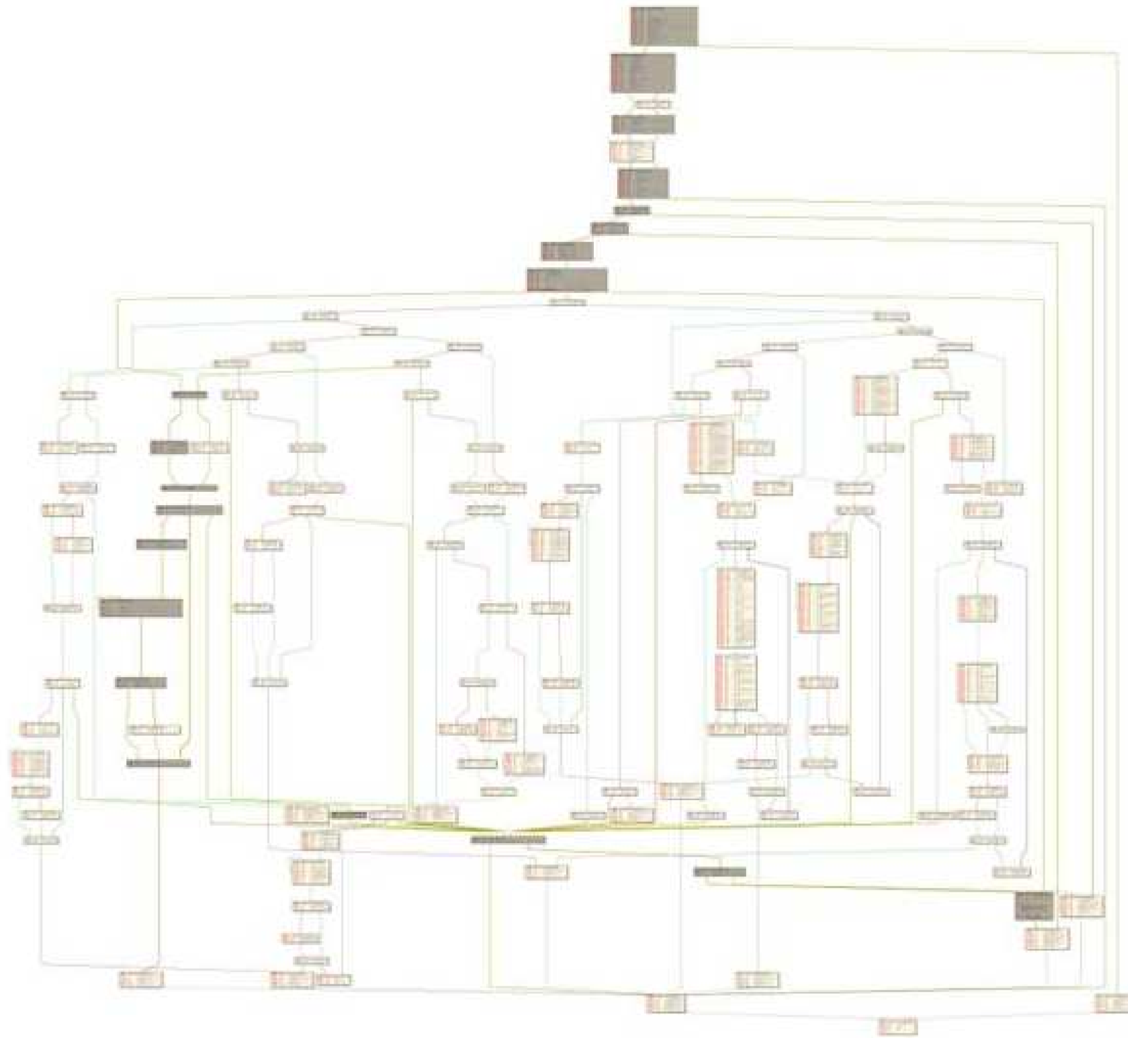


II. The trace/watch answer: Graph H' for **one specification** of $\{ a, b, c, d, e \} = \{ \alpha, \beta, \gamma, \delta, \varepsilon, \zeta \}$

Input tracking: enquiries

A: Quite simply: at often of times what the vulnerability analyzer wants is to *knowing its range of possibilities*. Possessing the dataflow φ to a given path in the program P has obvious advantages, but knowing this while being able to correlate such information to all possible code that use a given set of data greatly surpasses the first solution. So, focus at the relevant part and, then, determine the dataflow for one or another input and be able to build relationships between all possible input data-sets.

Input tracking: enquiries



III. Subgraph H of G intersected by H' (for **one specification** of $\{ a, b, c, d, e \} = \{ \alpha, \beta, \gamma, \delta, \varepsilon, \zeta \}$)

Input tracking: references

[1] Arthur Stoutchinin and Francois de Ferriere. Efficient static single assignment form for predication. In MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, pages 172–181, Washington, DC, USA, 2001. IEEE Computer Society.

Q&A

?