HBGary

Active Reversing

Greg Hoglund
Andrew Schaffer

The goal

Solve reverse engineering problems as quickly as possible without having to read disassembled code

Advantages

- Active Reversing reveals contextual relationships between user actions, behavior, code, and data
- Active Reversing excels at classification and sorting problems
- Active Reversing is really easy to use

The business case

- Active Reversing can save time – lots of time if used correctly
- Active Reversing increases the labor pool
  - People without disassembly skills can participate
- Active Reversing can be used in conjunction with traditional methods to increase productivity

The need for data

- Static analysis does not reveal data that is calculated at runtime nor does it illustrate motion – all of these things are left to assumption or prediction
- The need for data is the reason that even die-hard static reverse-engineers always drop into a debugger at some point, or perform real input testing

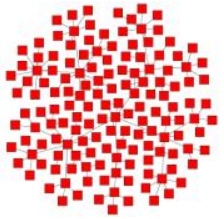
Here we are

- We present a new methodology that is very data-flow centric
- Our new method demands a whole new breed of tools
- We have prototyped several of these new tools and we illustrate how to use them
- Our company, HBGary, is committed to commericializing this new form of reverse engineering

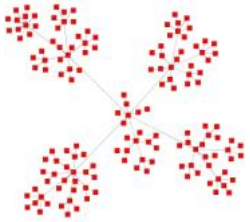
Part II

THE METHODOLOGY

The methodology



Code *and* data flow is harvested at runtime, collected into sets, and blended together into a graph…



...this graph is refined iteratively until it solves the reverse engineering problem.

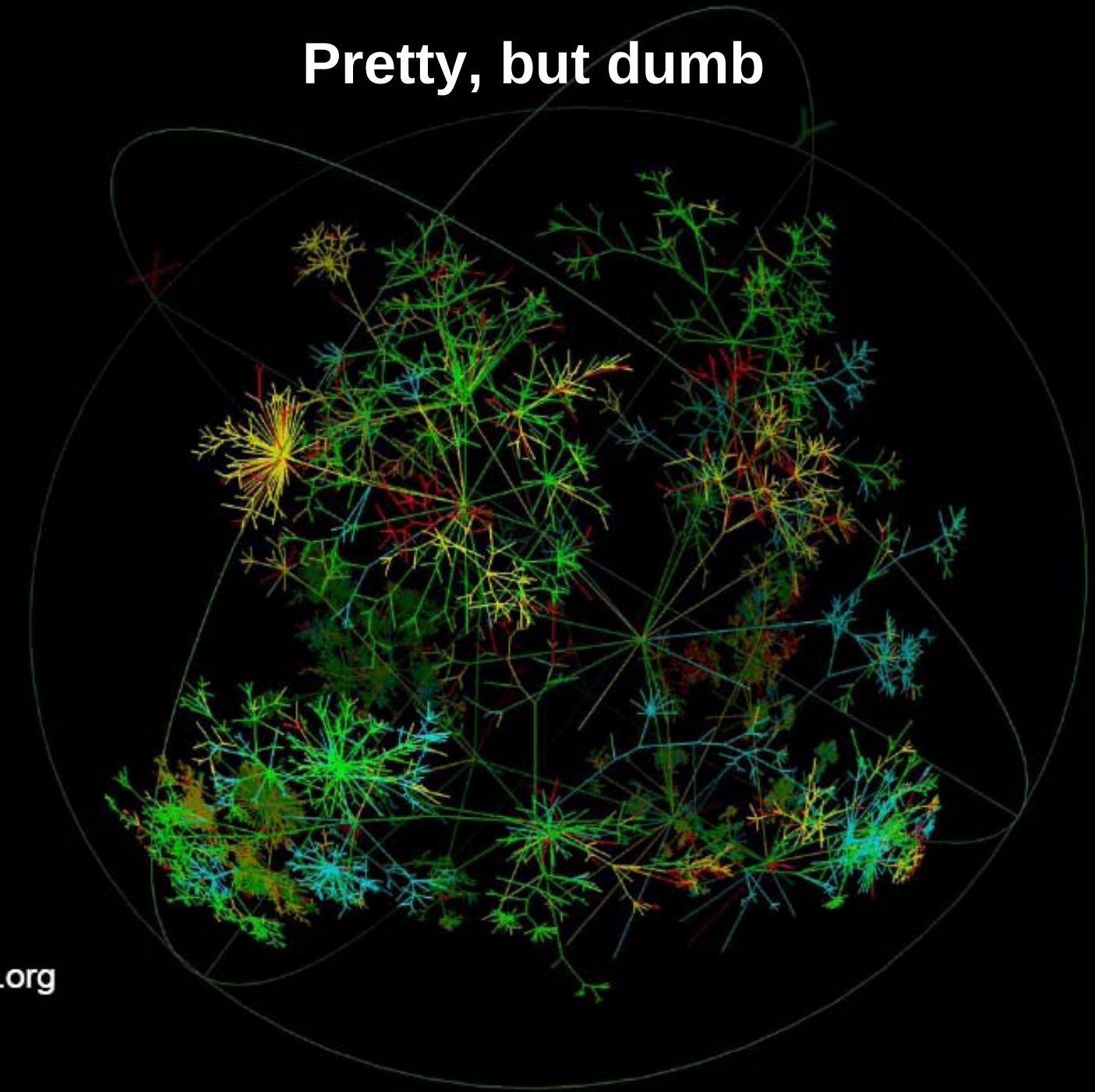
Yes, a graph

- Software is a bunch of small interrelated moving parts, naturally suited to a graph
- But, to work, the graph must be able to illustrate the solution data
  - Relationship between objects or events, membership in a particular set, presence of specific data or content, etc
  - Almost anything can be a node, and edges represent relationships between arbitrary things, so this is actually quite flexible

The “large graph problem”

- Historically, graphs have been too large to interact with
  - The key word is “interact”

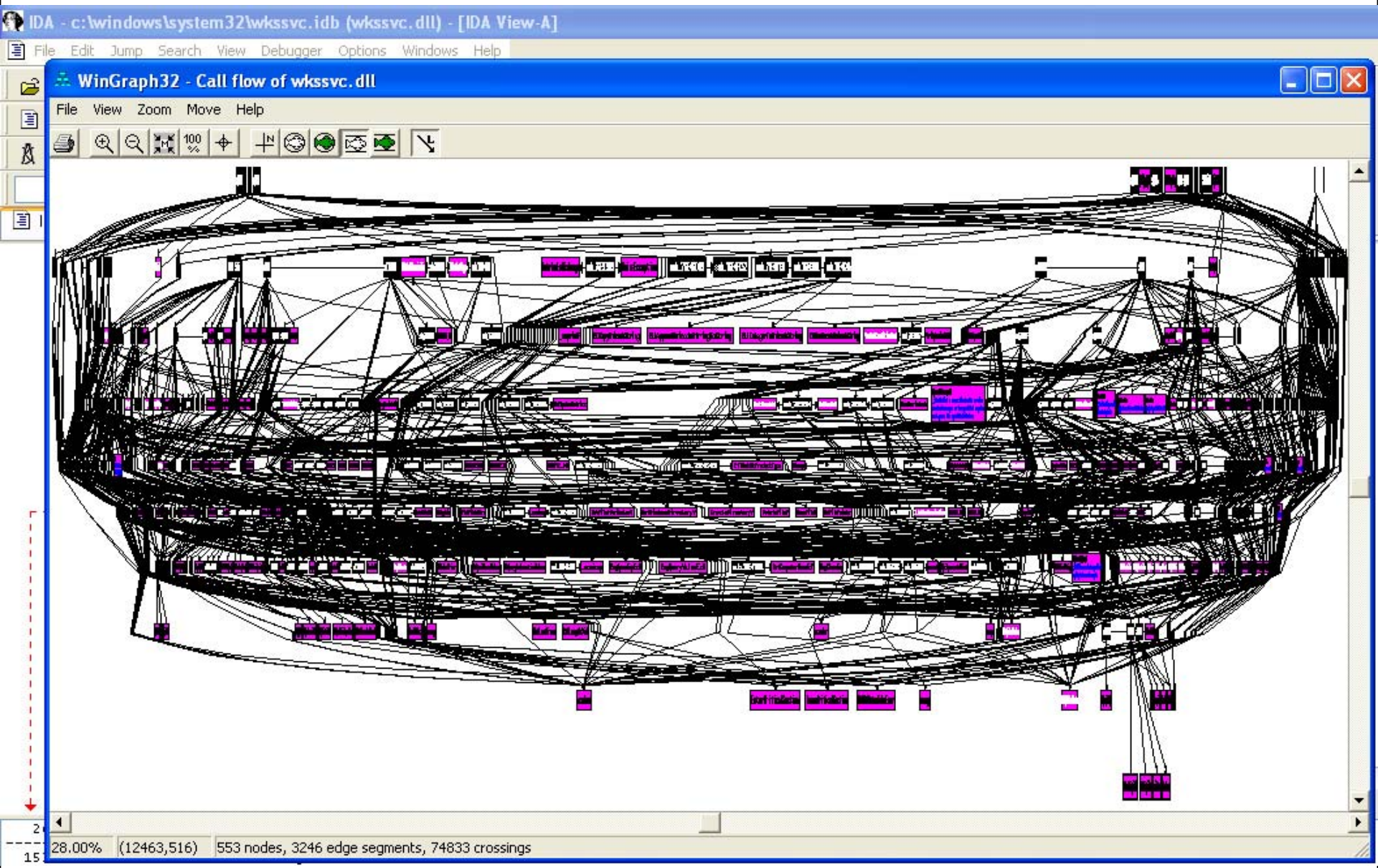
Pretty, but dumb



WALRUS

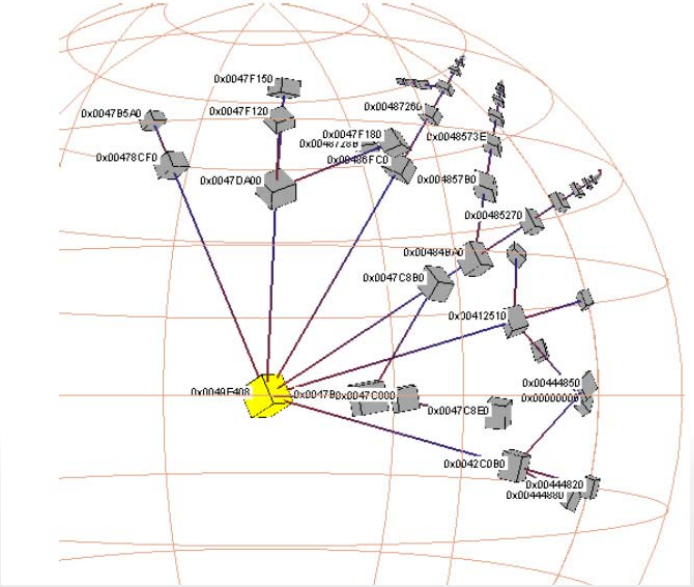
www.caida.org

Ugly, and dumb



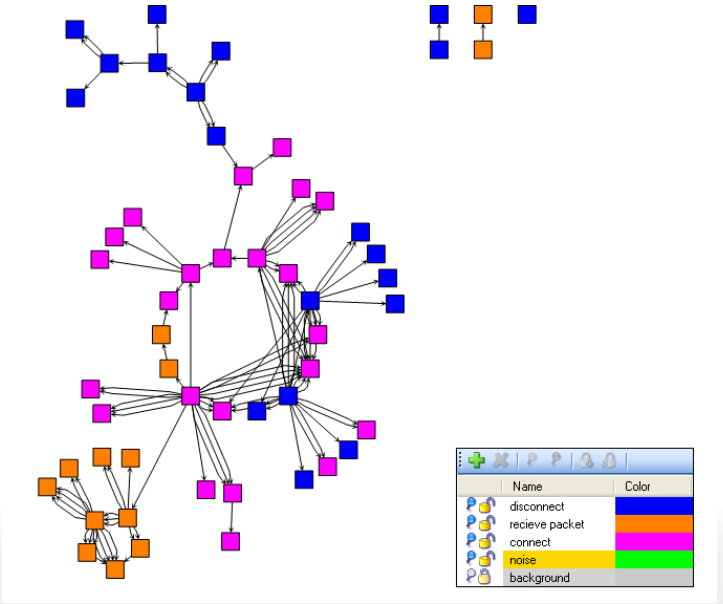
Hyperbolic Graphing

- Impressive and powerful, but not for us
- Designed for large directed graphs, but clumsy when dealing with smaller, more manageable sets



Stick to tradition

- Smaller, more manageable graphs are best drawn in the traditional 2D layout with color and annotations

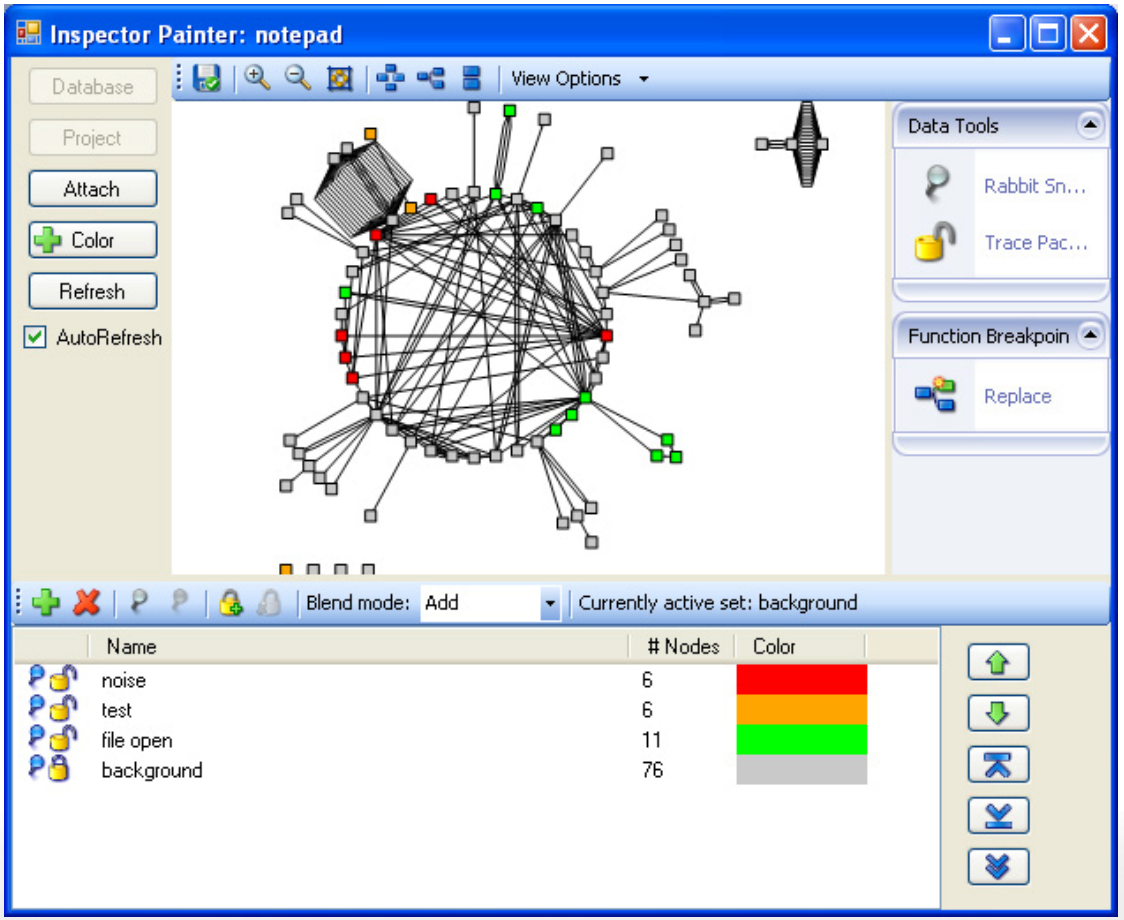


Data reduction and refinement

- The premise of Active Reversing is to show only what matters and nothing more
- There is a significant reduction in the amount of data that must be analyzed
- The refinement of the data converges upon the solution to the reverse engineering problem

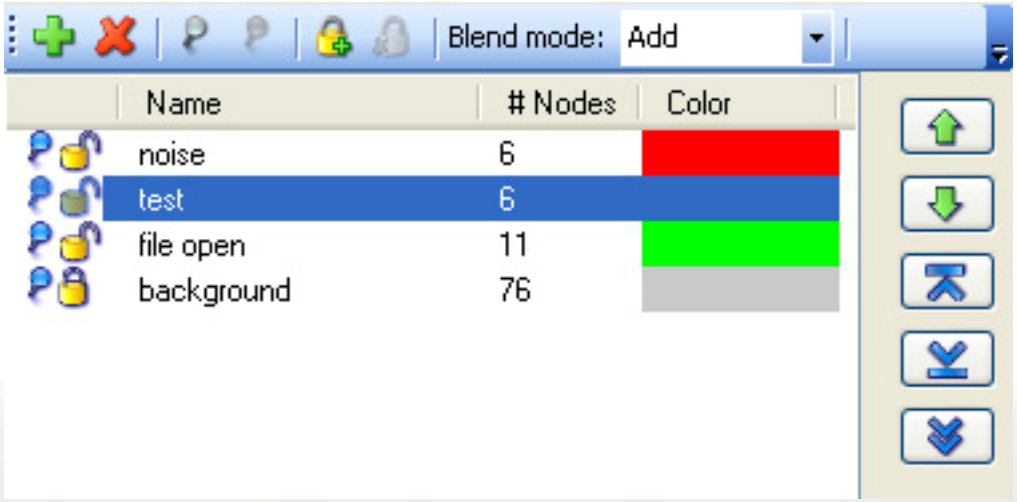
How: the working canvas

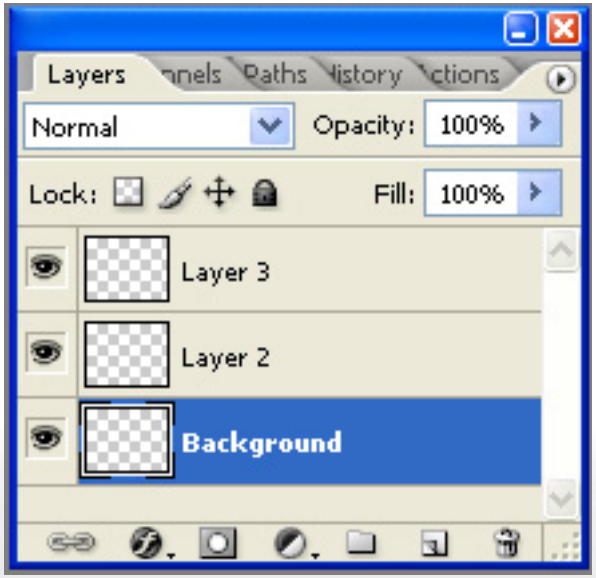
The primary workspace is known as the “working canvas”



Layers

- Sets are ***layered*** onto the canvas, much in the same way that layers in Photoshop™ are combined into an image





Set operations

- Layers are an easy and convenient way to combine sets
- All set operations (union, intersection, etc) can be represented using the layer system
  - … via order, visibility, and blending mode

Set harvesting

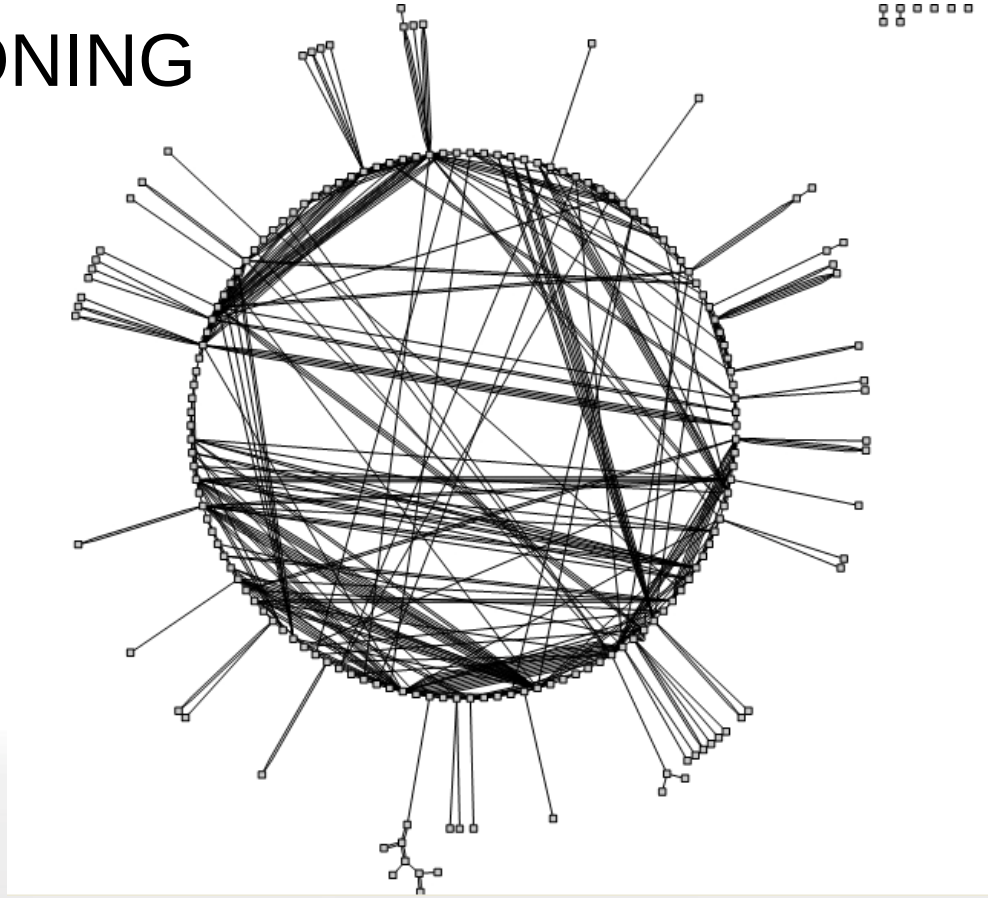
- We will cover many tools for set harvesting
  - Dataflow tracing
  - Hit counting
  - Function coverage
  - String references
  - Symbolic information

The methodology

Harvest, combine, and refine!

Part III

EXECUTION PARTITIONING



Active Reversing reveals contextual relationships between user actions, behavior, code, and data – to begin we start with code

Assumptions about Behavior

- Program behavior is in response to action that was just taken
- Different behaviors are represented by different code
  - This is how compilers build software

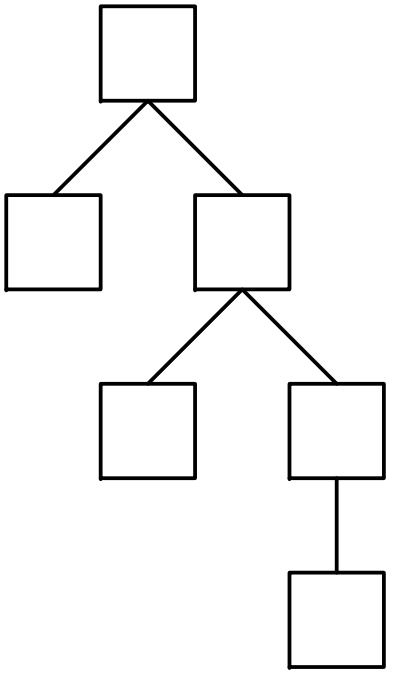
Examples of User Actions

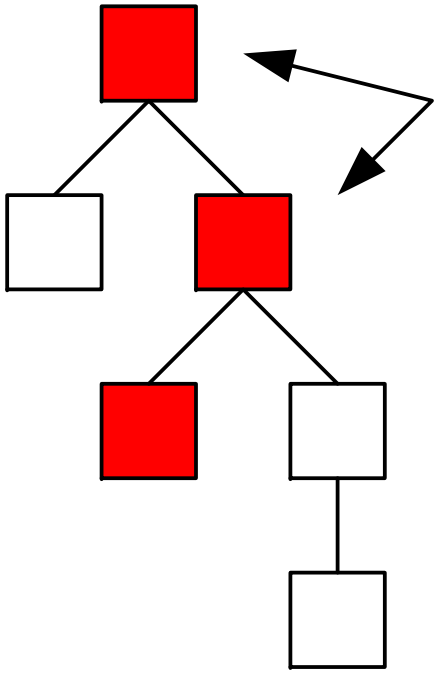
- Sending a packet
- Causing a specific transaction, such as a login or copy-file command
- Using a button or menu on the GUI
- Moving a game character in 3-space
- Unplugging or inserting hardware

Execution Partitioning 101

- Rapidly locate the function(s) responsible for a particular program feature, isolate code by functionality
  - Incremental coverage sets
  - Noise removal

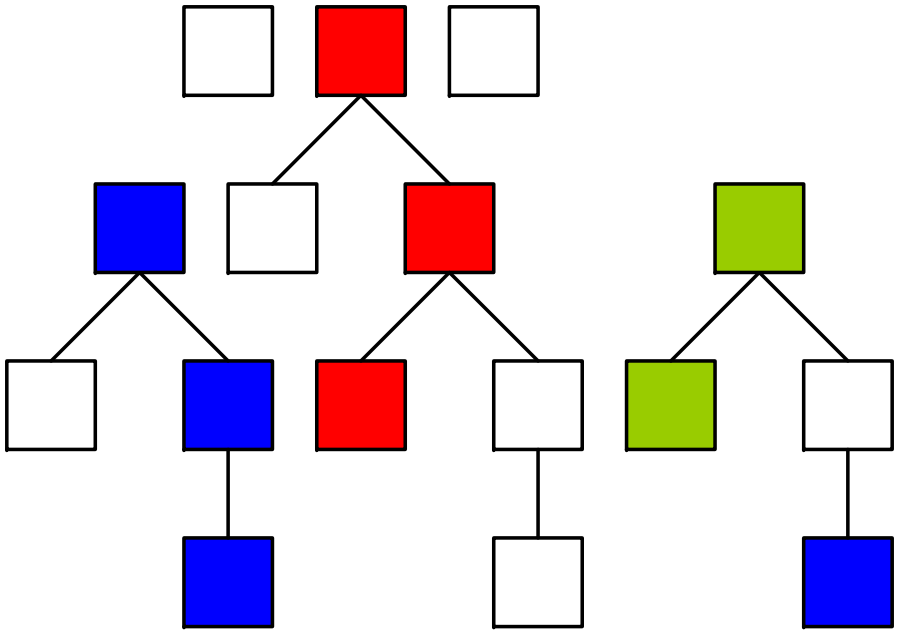
Function Coverage





These functions have executed

Partitioning



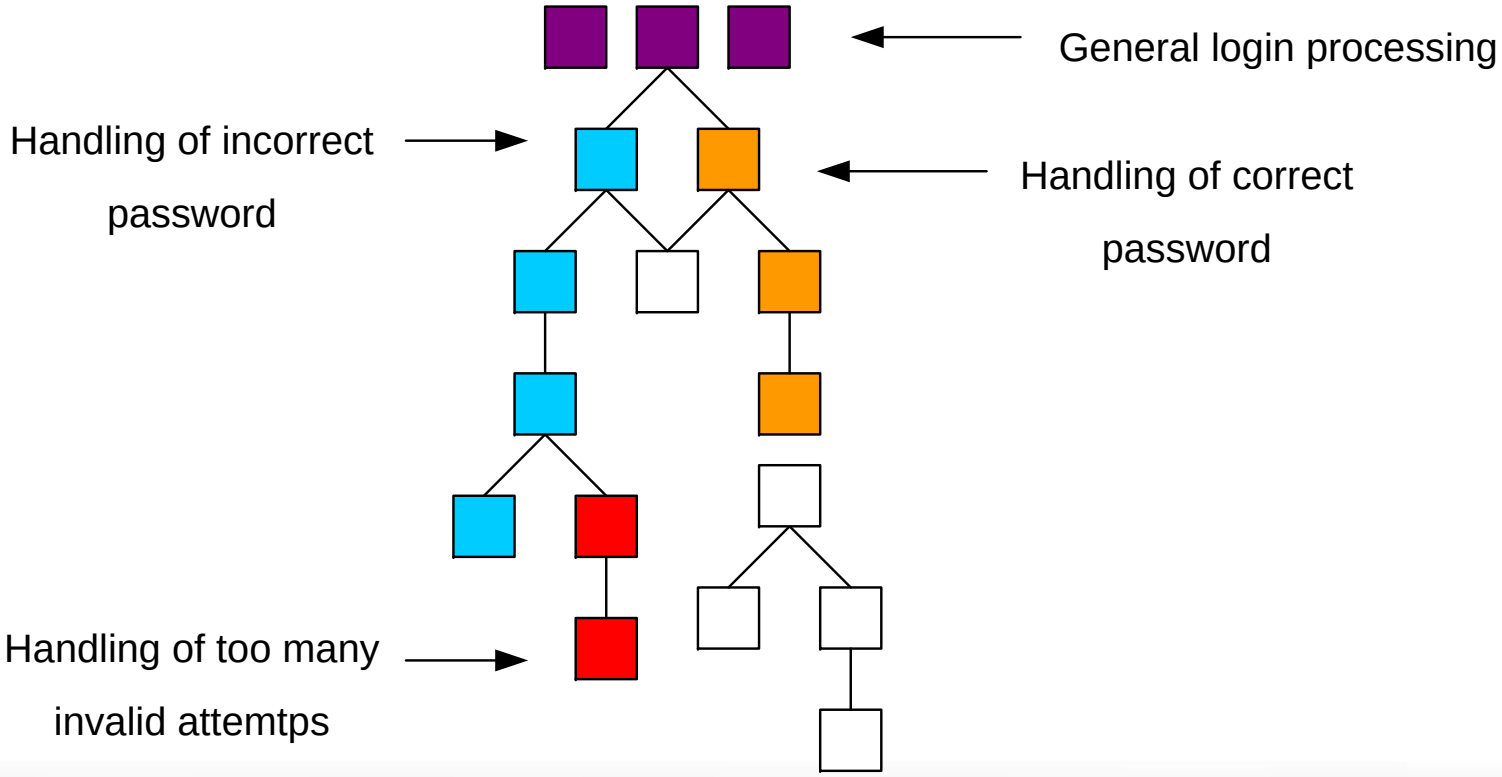
Execution Partitioning 201

- Change up the data content of the transaction to induce many possible responses

Remember the data too!

- Its not just about packets and menu items, but also about the data you type or insert
- The contextual data associated w/ the user initiated action plays a large part in how the program logic will respond
  - A packet w/ a bad checksum won't get far
  - '$%%%%%$$$$$' in the file-open dialog will do something different than 'aZAzazzazAA'

Partitioning more detail



Example: File Paths

- TBD

Execution Partitioning 301

- Force error conditions, abortive logic, and exceptions through both data and direct action

Remember the error state

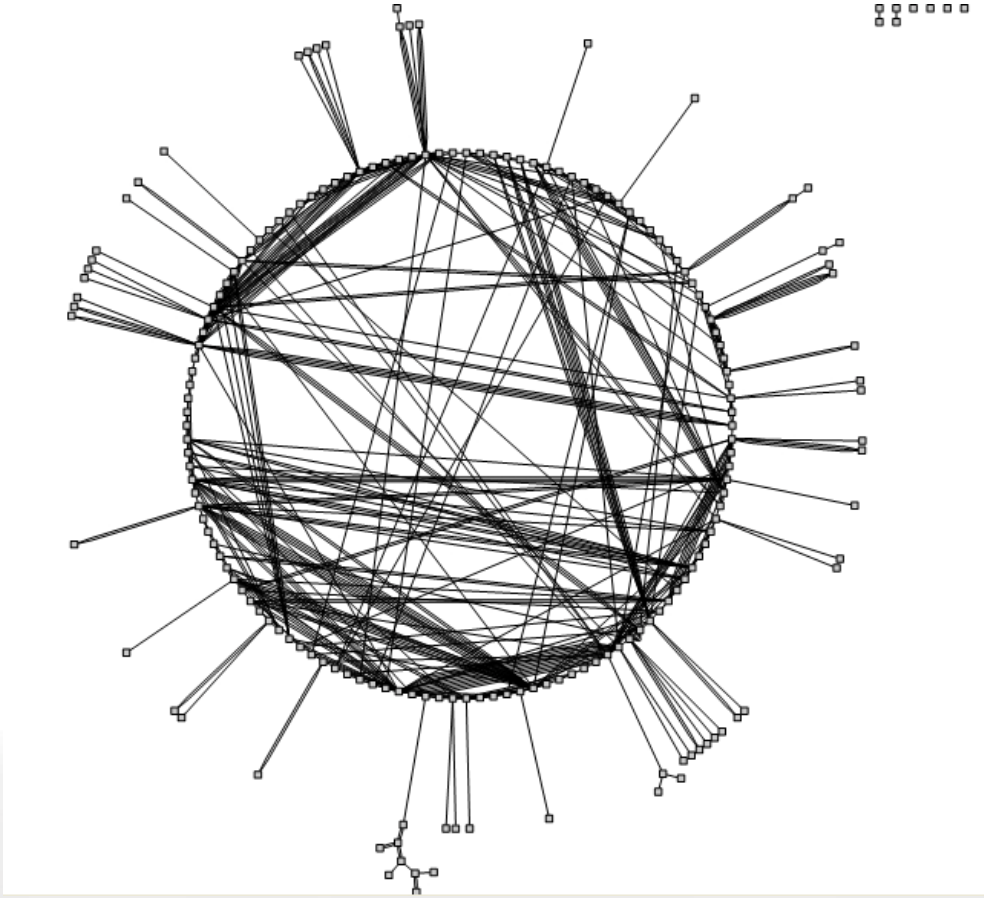
- Many user-initiated actions can induce both success and failure logic
- Sending a good password verses sending a bad password
- Moving before the spell-casting is complete
- Unplugging the network cable when a file transfer is in progress

Example: bad login

- Response to bad password will cause some error handler to execute
- Response to good password will execute a whole series of connection-initialization routines
- The code for these two responses are physically separated in the program code

Part IV

DATA SAMPLING



Active Reversing reveals contextual relationships between user actions, behavior, code, and data – now that we have code we can move on to data…

Assumption: Data follows code

- It makes sense that code that implements behavior must also touch data related to that behavior
- Code and data flows are tightly coupled
- They co-exist spatially in the context of the stack and the CPU registers

Where we are

- At this point in the process, your graph should be well partitioned
- Because we know data follows code, we can begin examining dataflow by going to the already existing partition of interest

Data sampling

- Collect a detailed instruction-by-instruction sample history for a defined region of code
  - The collection space is bounded by the partition set thus granting a manageable computational overhead

Example: looking for SQL statements

- Find a region of code that is related to login
- See if you can recover the SQL statements

Data sample searching

- Specific value search
  - You must know the specific value ahead of time
    - Can you query it from the software? (XYZ coordinate?)
- Use regular expressions to perform detailed pattern scans over the sample set
  - Allows much larger sample sets to be analyzed in much shorter time if you already know what you're looking for

Example: searching

- Perform SQL search… TBD

Tool: Data taps

Tool: Statistical analysis on value series

- Packet types over time

Tool: Conditional triggers

- Trigger a deep trace on a specific data state and control flow location
- Extends an existing partition, or builds a new partition by leveraging an existing one as a ‘jump off point’

Example: Give me Warden!

- Capture all the instructions of the warden client
  - Conditional deep trace on packet type (2E8?)
  - Add new functions into new set
    - Avoid adding functions from system DLLs

Proximity Relevance

- Cluster functions by relevance to a buffer or other memory range
  - Good for class reconstruction

Locate the allocate and copy routines in the MIME decoding class

- I need the allocation and copy routines so I can locate potential buffer overflows…

Freeform memory scanning

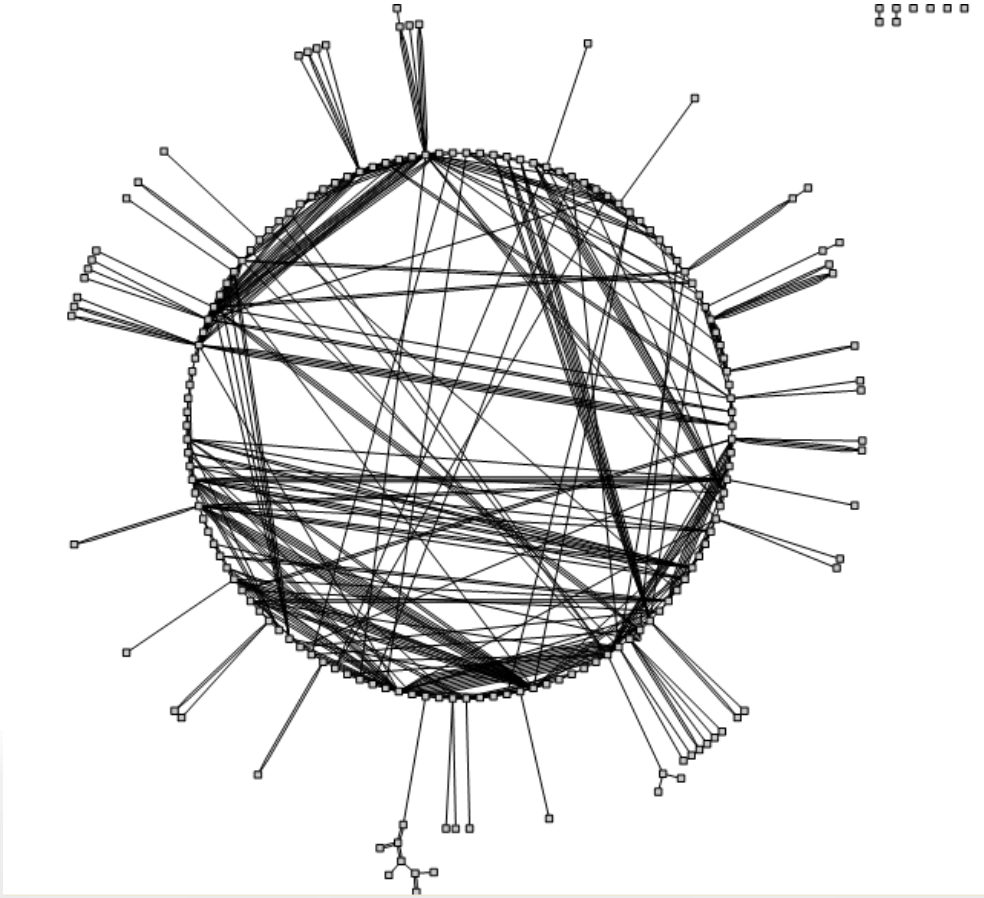
- Scan all of memory for a value
- Use hardware breakpoints to break on access
  - Limited to 4 at a time
  - Avoid stack addresses as they are constant flux
- Works well when you don’t have a well partitioned starting space

Example: Finding the code that generates the login packets for WoW…

- I need to find the login function for this game so I can build an emulation server…
  - Rabbit snare the login name
  - Dataflow trace
  - User-determined execution partitioning
    - which functions execute when we log in

Part IV

DATAFLOW TRACING



Dataflow

- Trace every instruction and record how it effected the data
- Trace all propagation of data
- Record the arithmetic transformation at the time of propagation
- View the transformation history on any data instance

Functions use derived values and copies

- In many cases, functions deal with copies of the original data, or values that were derived from the original data, so tracking just the initial memory range is not enough
- Dataflow tracing reveals many more functions that deal with the subsequent data

Tool: Follow a buffer

- Follow a buffer, such as a packet, to track all derived values and copies of values that propagate into the program and reveal any function that touches any of these derived values

Example: Find the telehack kickme!

- Locate part in WoW that kicks you offline when you alter XYZ coordinates…

Tracing reveals arithmetic

- When data is moved into a register, it can be traced against all operands that use this register
- All instructions that perform comparisons or calculate results from the data can be recovered
- All conditional branches based on the data can be recovered

Tool: Class member type recovery

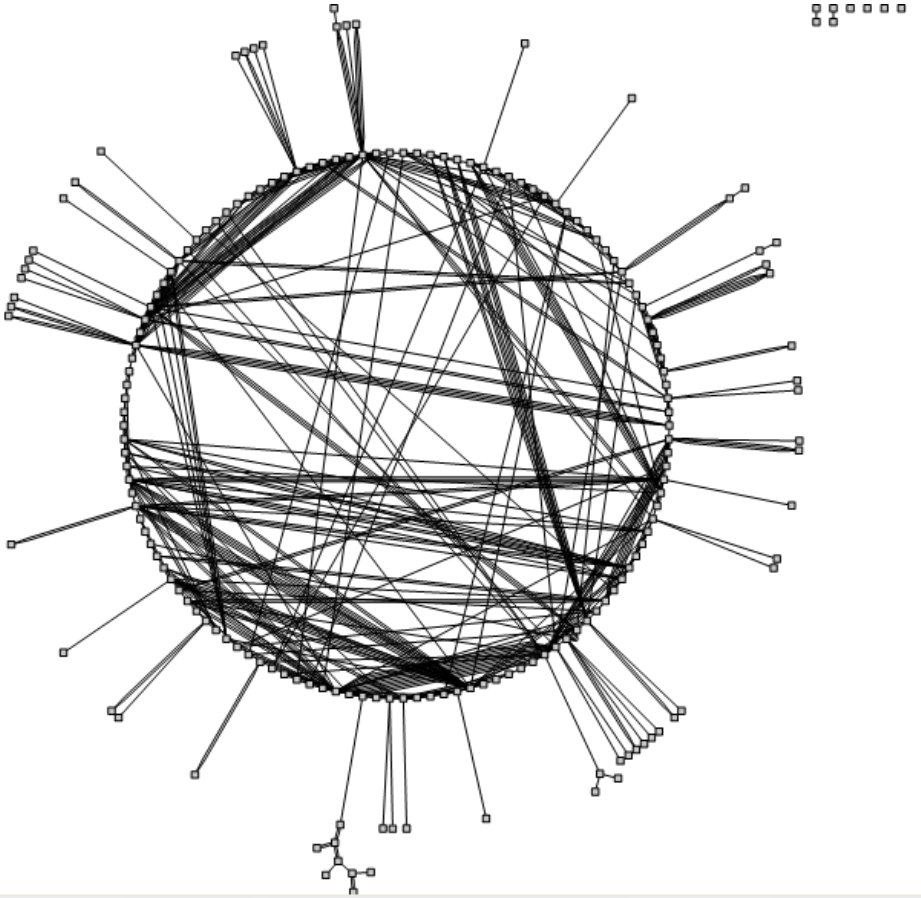
- Once offsets into a class instance are detected as being loaded into a register or temporary variable, dataflow can be used to trace out all arithmetic operations on that data
  - Signed / unsigned, char, short, long, float, etc.

Example: reconstruction of player structure in Lord of the Rings Online

- TBD

Part V

HITCOUNTS and SEQUENTIAL SET REDUCTION



Hitcounts

- Cross reference known quantities of events with meta-data collected
  - Number of accesses
  - Number of times executed

Sequential reduction

- Iteratively generate unique results and filter out nodes that do not accurately track the sequence
  - Removes temporary values
  - Removes false positives
  - Removes values from high-flux memory regions, such as the stack

Comparative memory scanning

- Compare multiple sample sets at known value states
- Iteratively filter non-tracking values
  - Increase
  - Decrease
  - Increase by #
  - Decrease by #
  - Change to #

Example: Finding XYZ

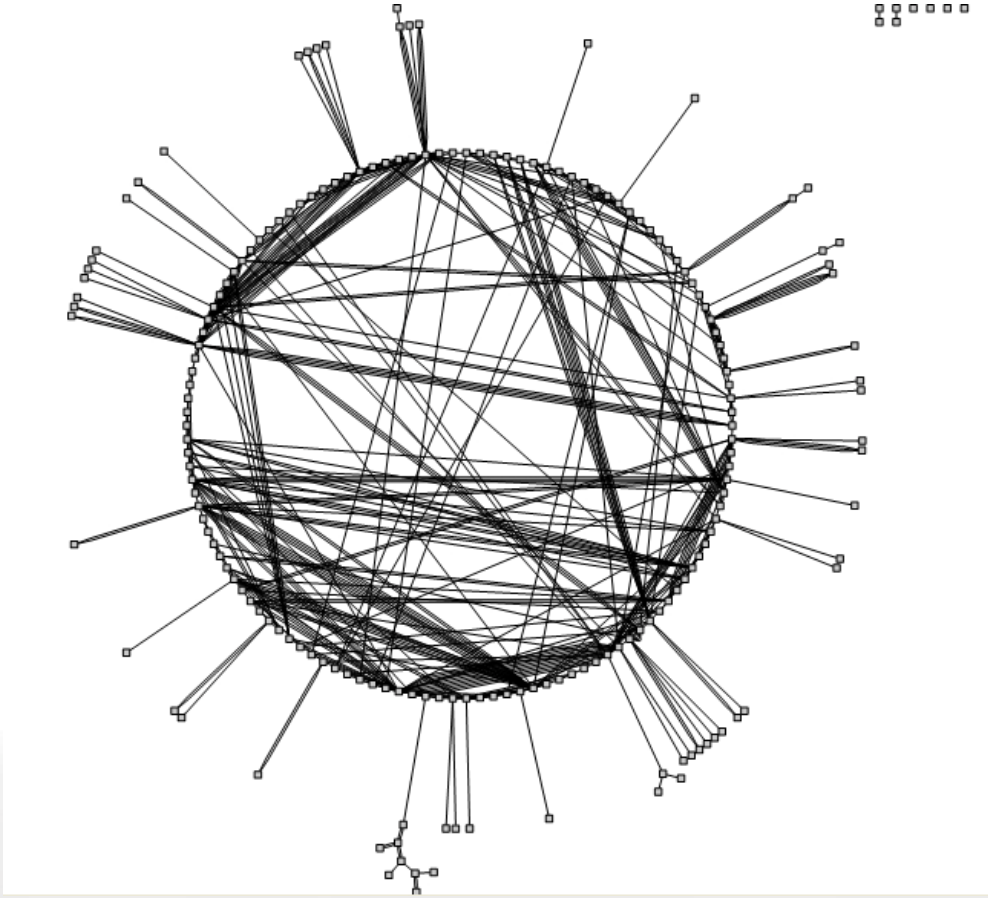
- I need to find the XYZ coordinates of this game character so I can test out telehacking….
  - Comparative memory scanning
  - Observable Reaction assessment

Hitcount extrapolation

- Compare hitcounts against multiples of a known quantity of events
  - User-induced events

Part VI

DENSITY ANALYSIS



Instruction Density Analysis

- Instruction-type density (e.g. FPU, arithmetic, MMX, etc..))
  - Used to classify functions based on what types of operations they perform.

Example…

- Isolate the code that handles 3D rendering
- TBD

Example

- Isolate the code that handles cryptography
- TBD

Exit/Throw Density Analysis

- Identifies code that handles parsing or is for some other reason filled with checks.

Help me build a better fuzzer!

- Useful in fuzzing or other vulnerability analysis.

Memory Range Density Analysis

- Accesses to a specific address or address range
  - Access density in this case can be used as a metric of degree of relevance to a given buffer.

Sort the packet parsing functions…

- The main packet parsing function for instance will access the packet buffer several times, but other functions might access small pieces of it(like the packet header, to figure out which parser to send it to). This way these two different types of functions can be differentiated easily.

Value Instance Density Analysis

- Accesses/appearances of a specific value or value range
  - » Highlight code relevant to a specific value or set of values, such as an input string or number.

Branch Density Analysis

- Branch density is a useful metric of the general complexity of a function.

Find the Big Fat Parser!

- When looking for a large parsing routine, filtering out smaller less complex functions would be helpful.

Hitcount Density Analysis

- Execution density (e.g. number of hits on a given function

Known argument counts

- Filter your set with more specific CXCCC

Operation-specific memory/value manipulation

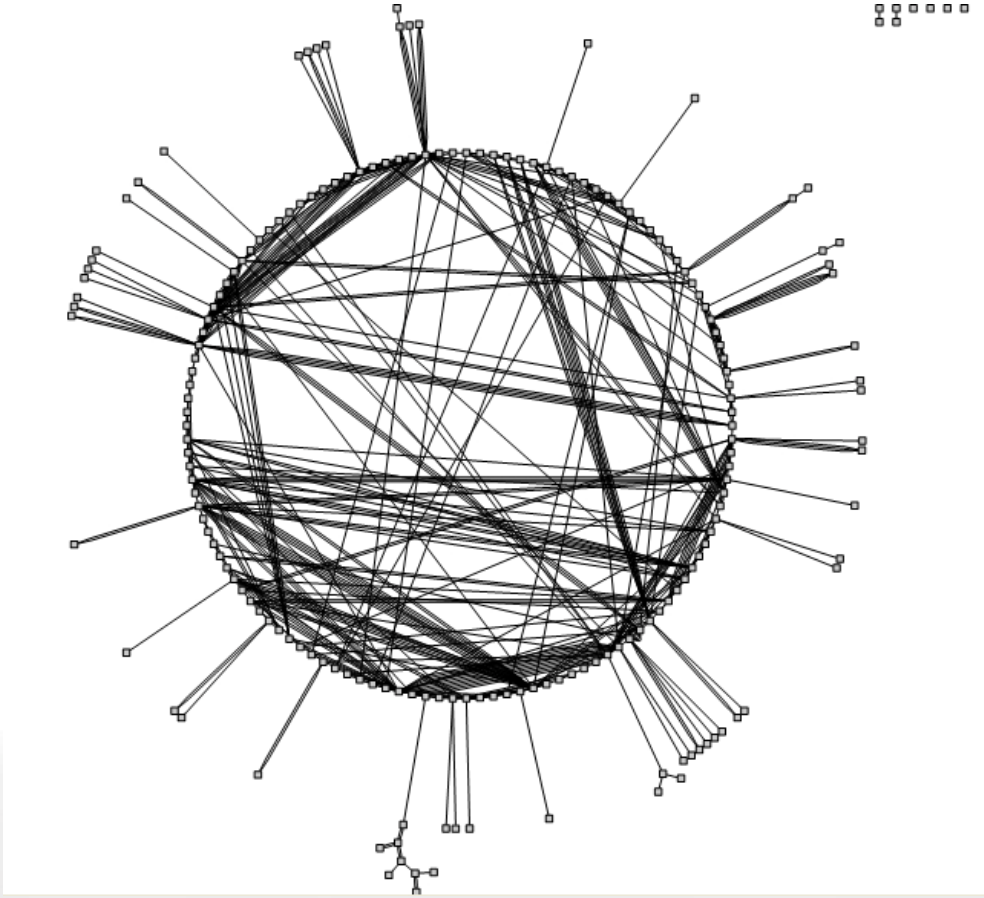
- Filter your set with more specific CXCCC

Loopcount Density Analysis

- Filter your set by DDDDD

Part VII

AUTOMATED FLOW RESOLUTION



Starting with dataflow

- Because we have the ability to collect all arithmetic that influences branches, we have the ability build equations that represent what is required to change those branching conditions
- We can calculate the required inputs to change the branch

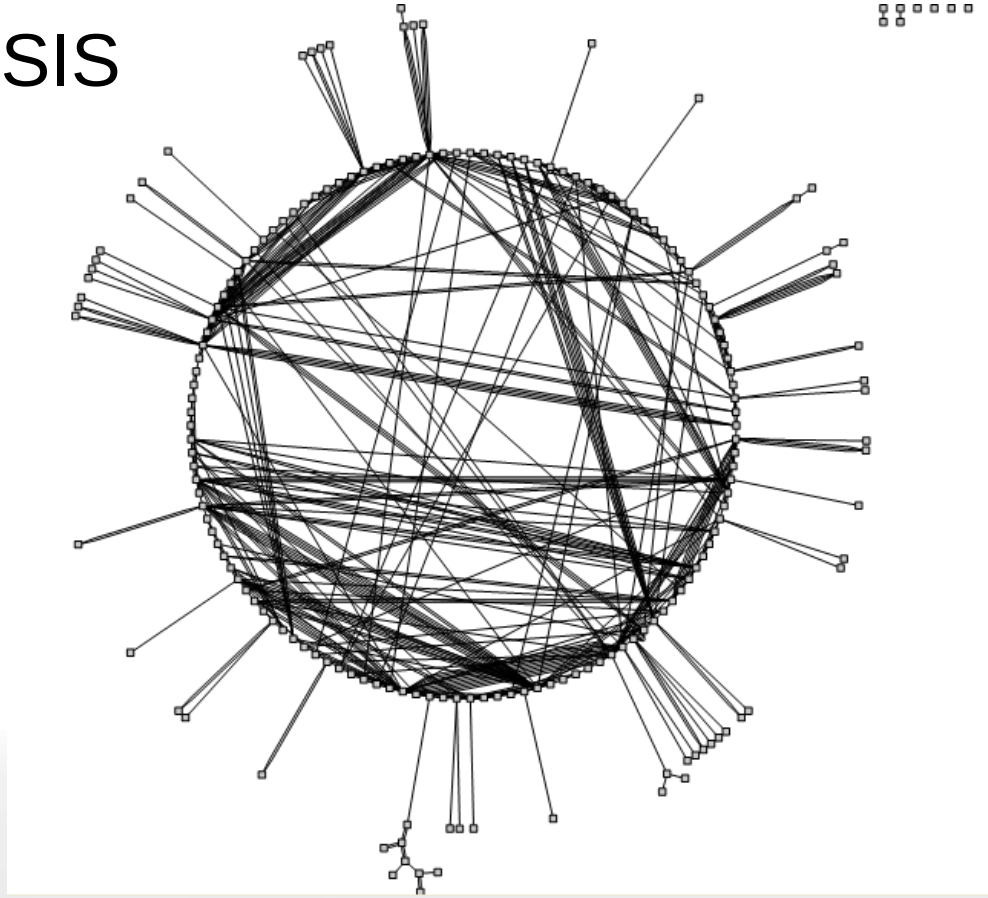
- TBD

Goodbye fuzzing!

- Input tests no longer need to be random, they can be derived from calculations against the dataflow set
- Of course, come calculations cannot be reversed, such as a hashing function, but the majority of software branches are a result of direct value comparisons or simplistic, reversible arithmetic (such as parsers)

Part VIII

PHASE SPACE ANALYSIS



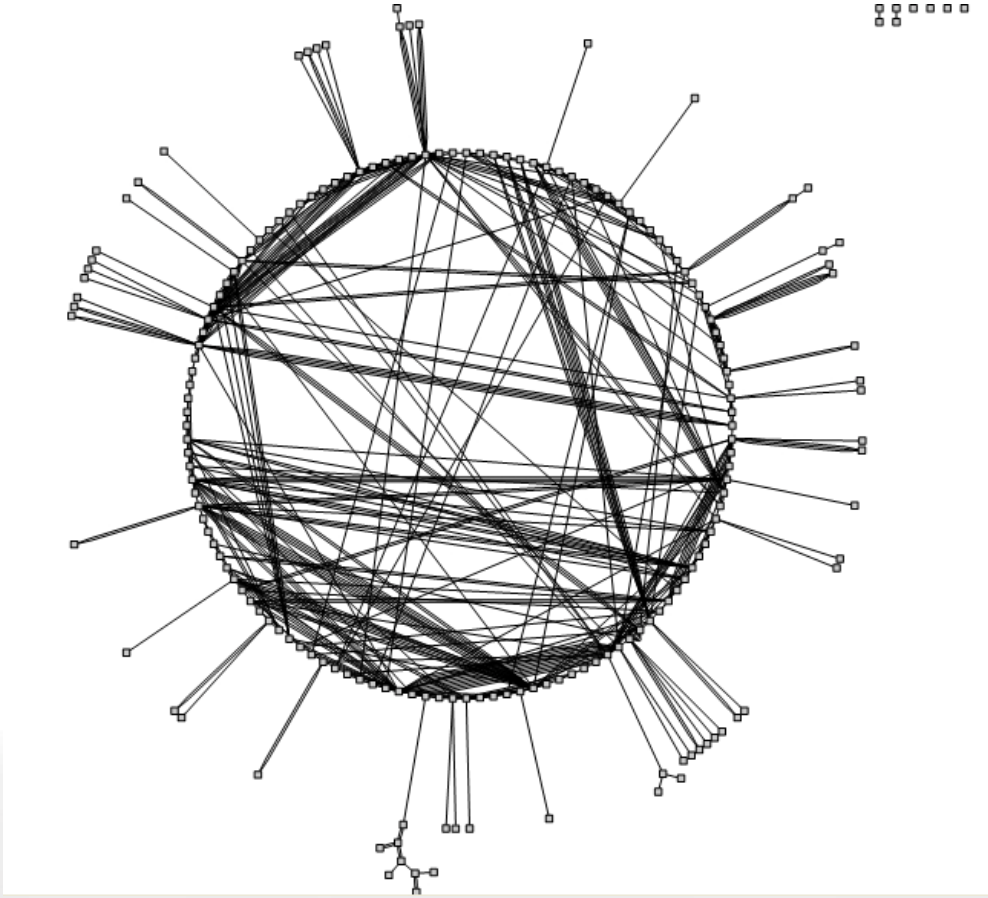
Phase Space Analysis

- Determine if program behaves significantly different than the baseline
- In conjunction w/ fuzzing, input tests, etc to determine if a new unique behavior has been uncovered
- Phase-sets are a more advanced form of set-analysis

EXAMPLE – SQL Injection

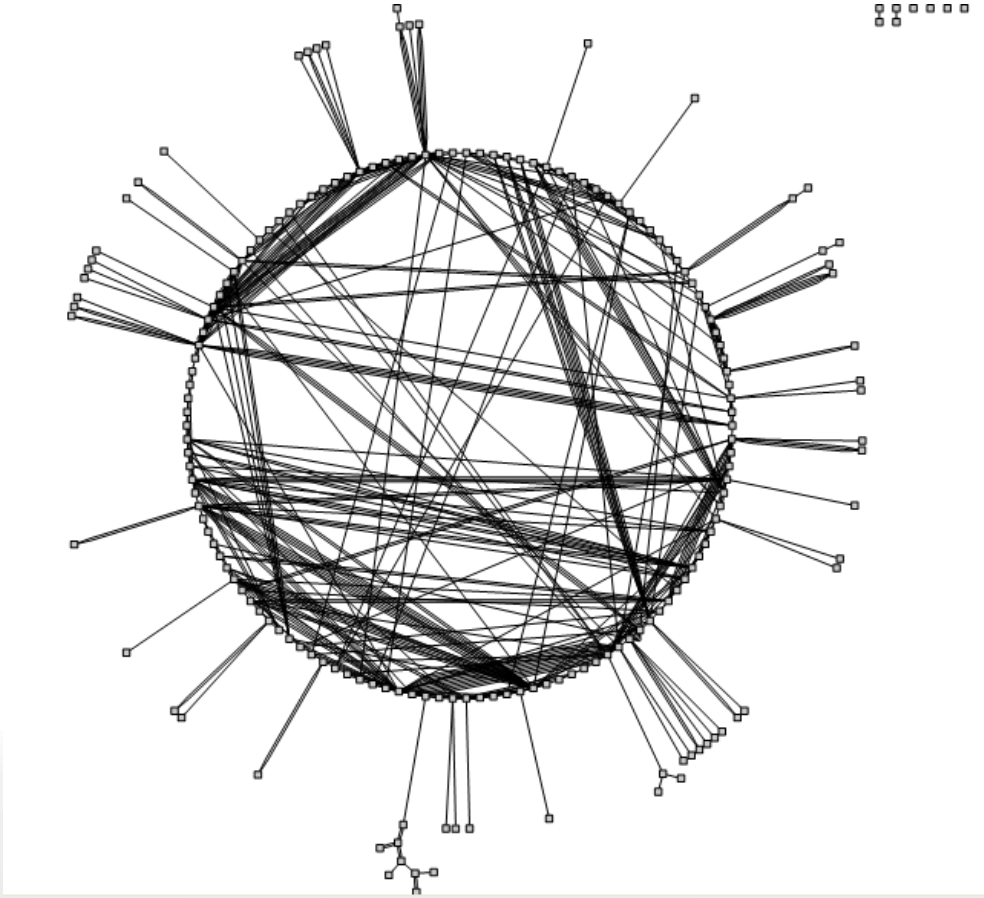
- Use SQL metacharacters causing a distinctively different reaction

Futures



- TBD

Conclusion



- TBD

HBGary

- www.hbgary.com
- hoglund@hbgary.com
- andrew@hbgary.com