Taking it From The Source

Brought To You By....

This is a research project conducted under the Defense Advanced Research Projects Agency (DARPA) Cyber Fast Track program

PI: Jared Carlson
Principal
GoToTheBoard

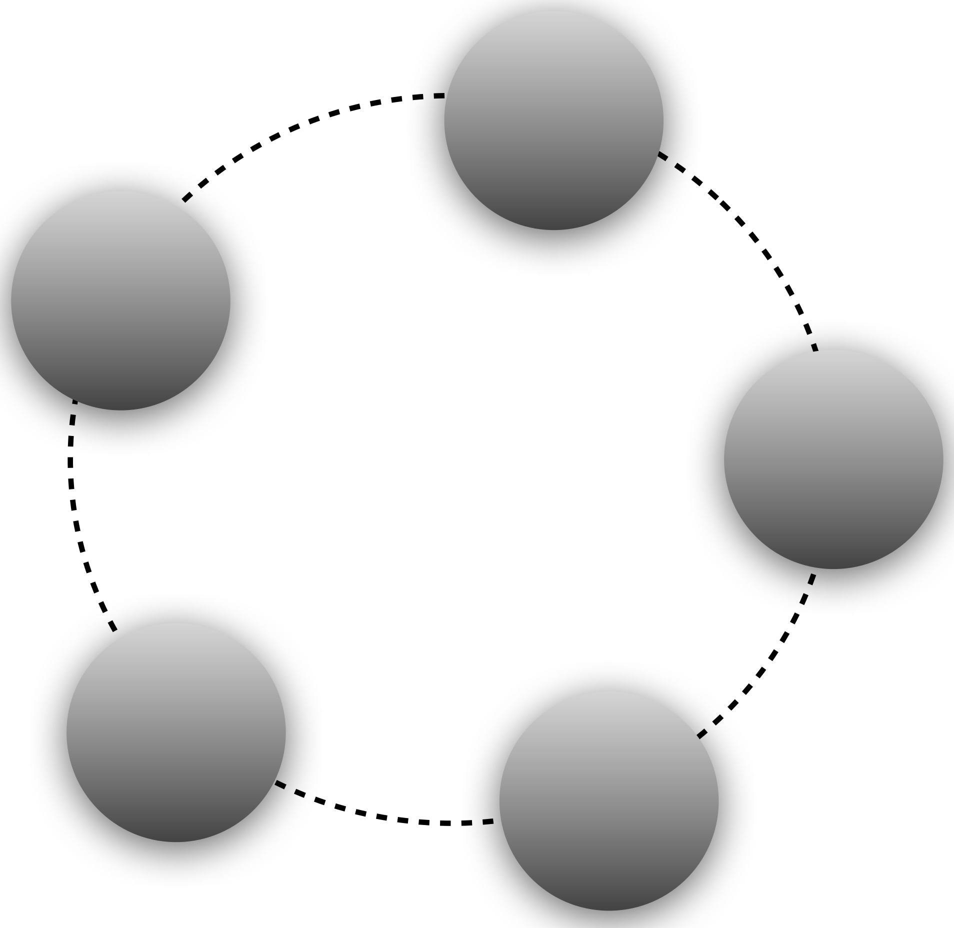
Disclaimer

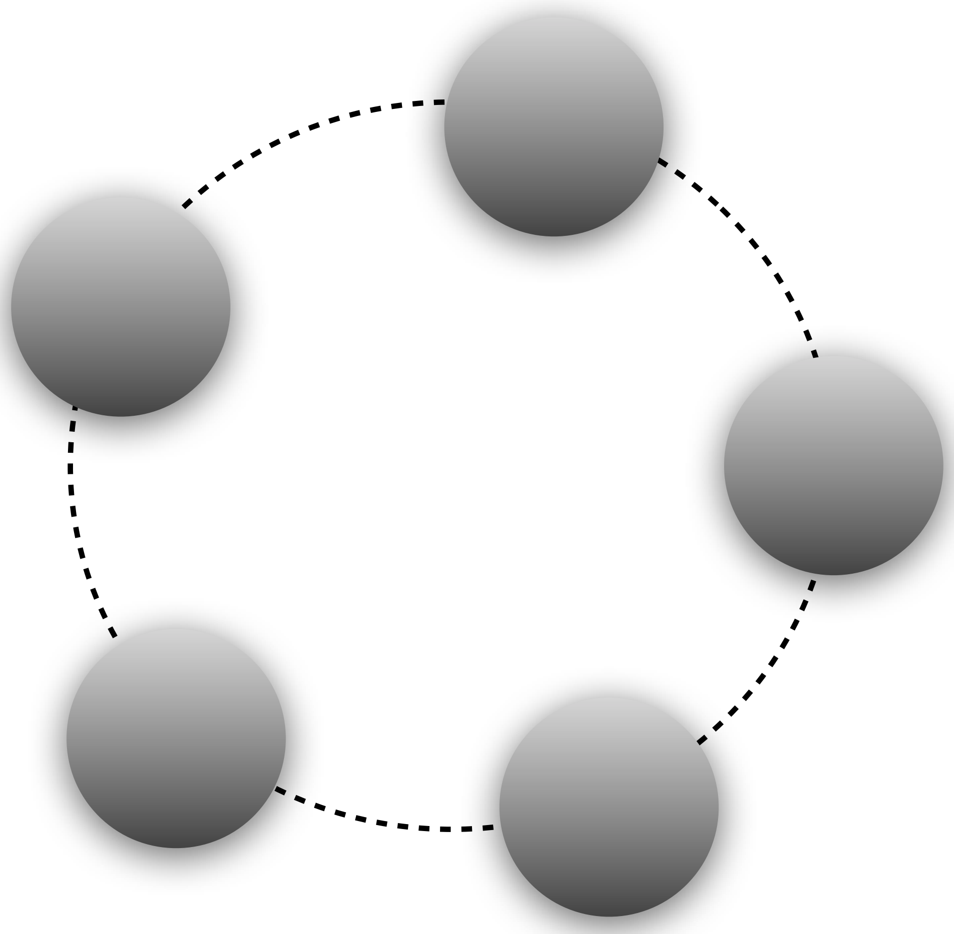
A Few Other Notes

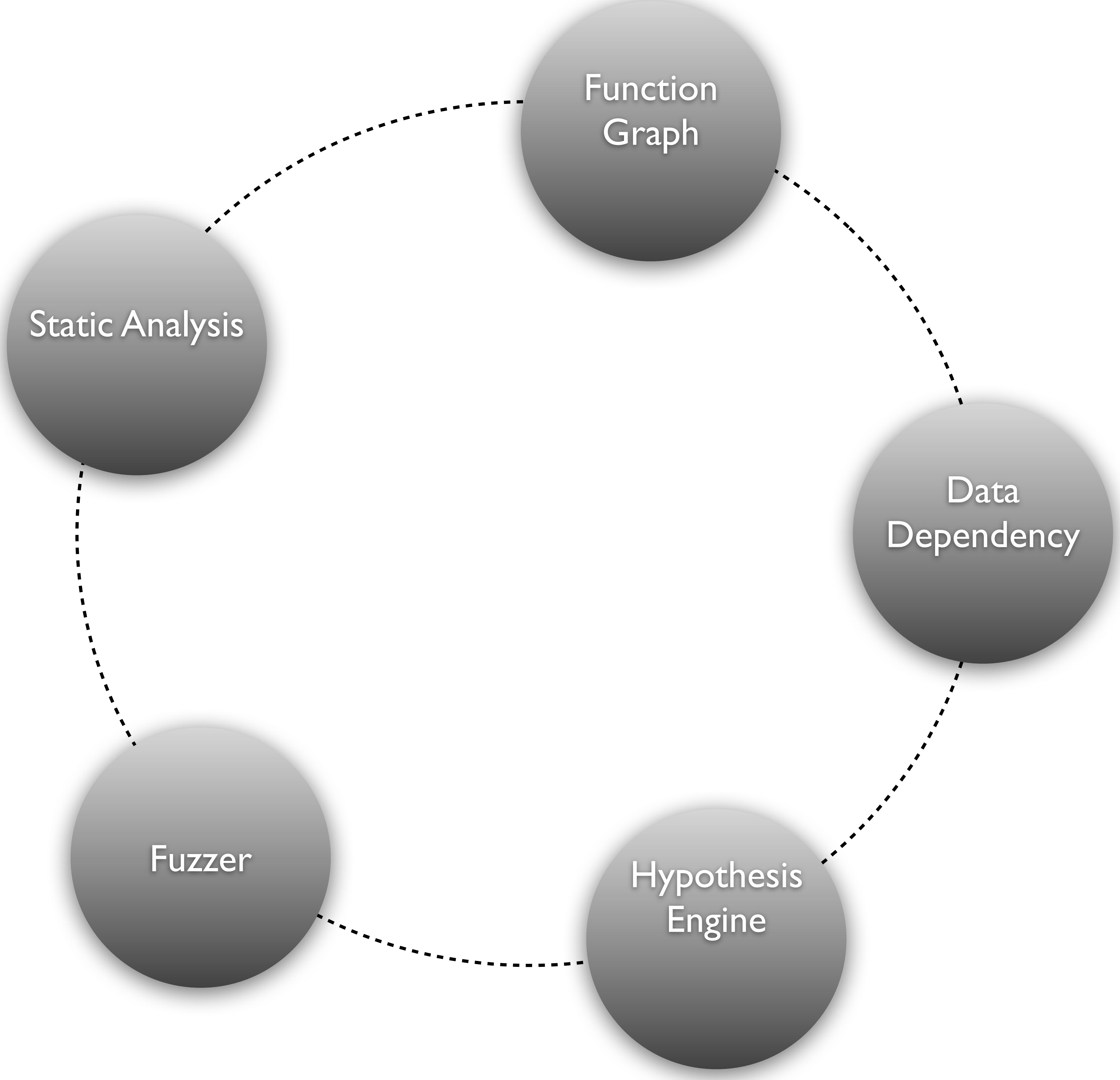
- This project is ongoing and this presentation is subject to a public release
- This means material is a little older than what’s actually in development.

Project Goals

- Create a tool to find exploitable bugs within a "normal" environment.
- Illustrate consequences of these bugs.
- Educate and interact with the developer.
- Allow for community improvement and sharing.







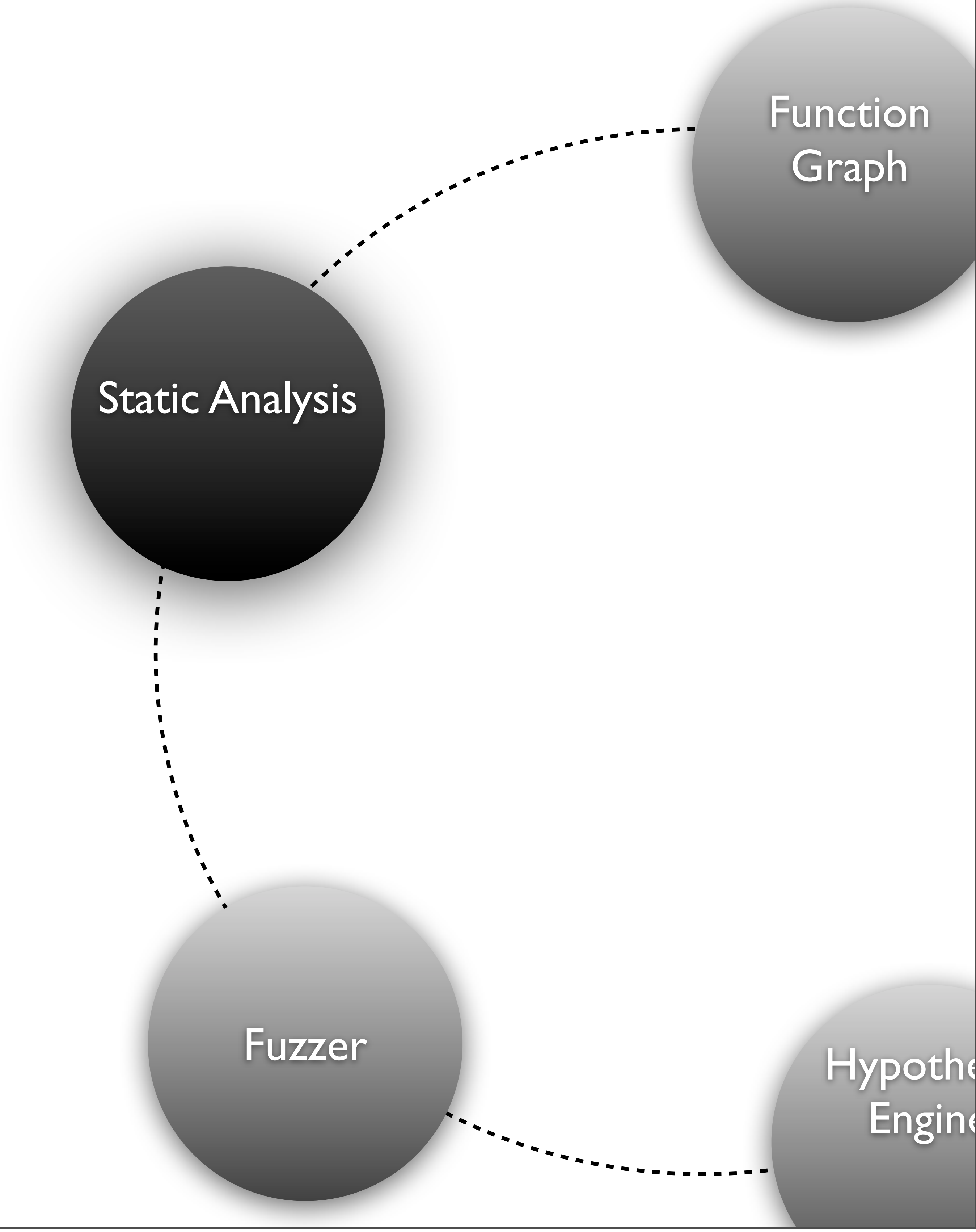
Function Graph

Data Dependency

Hypothesis Engine

Fuzzer

Static Analysis



Function Graph

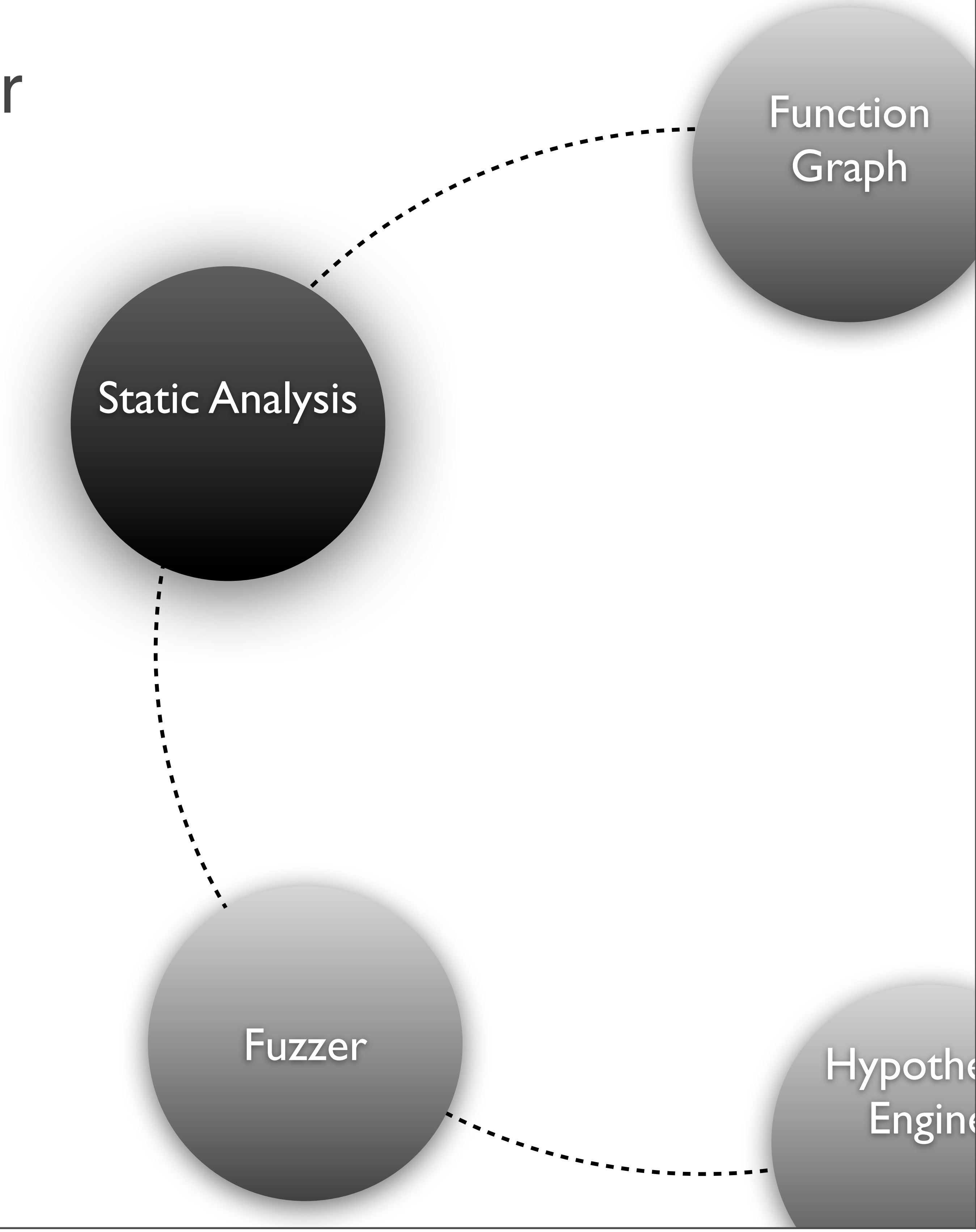
Static Analysis

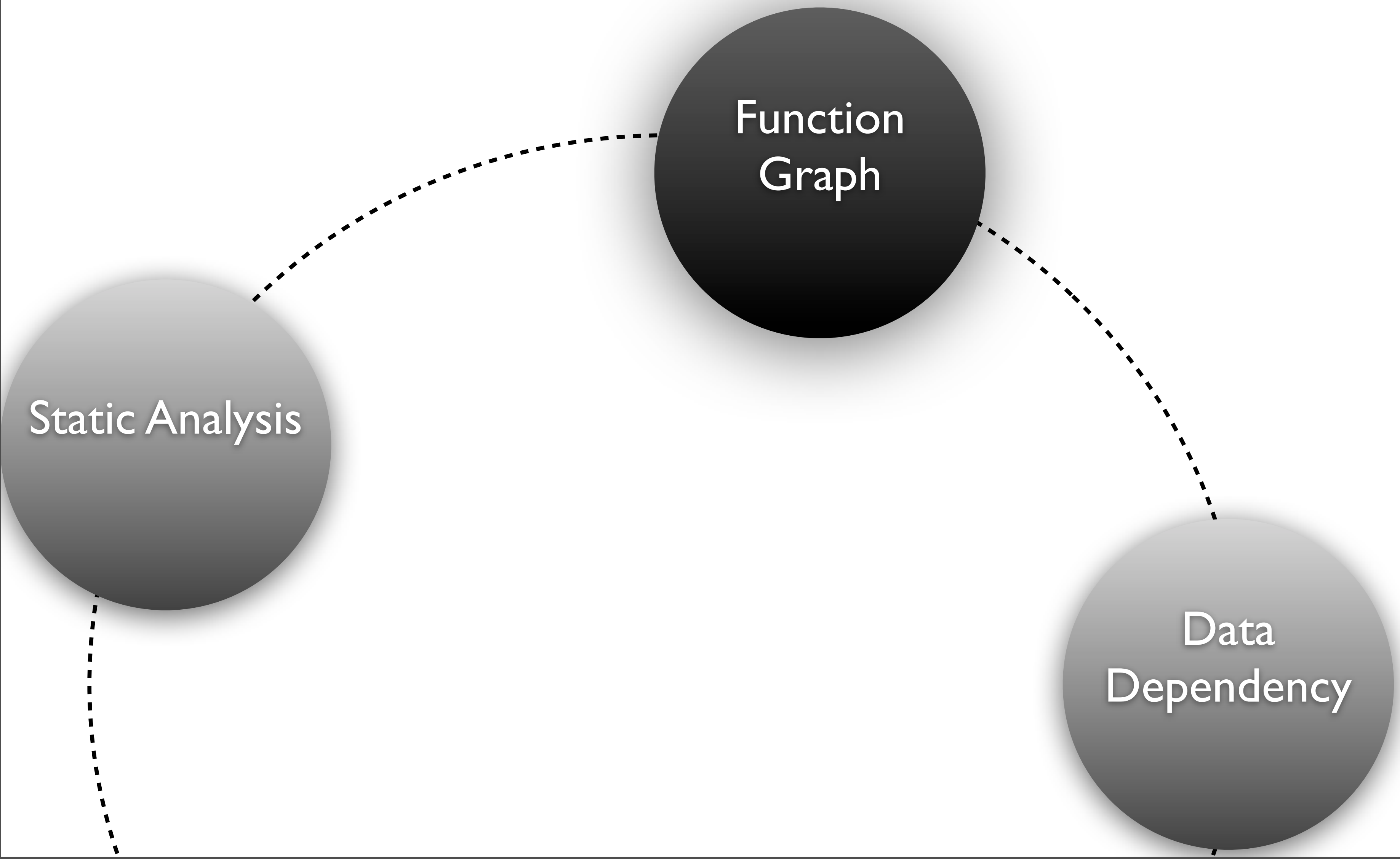
Fuzzer

Hypothe Engine

Static Analysis

- Rewrite scan-build/analyzer in python for integration purposes
- Digest & Export JSON rather than HTML for comms
- Reuse existing infrastructure and passes for more exhaustive analysis
- Expand using LLVM passes





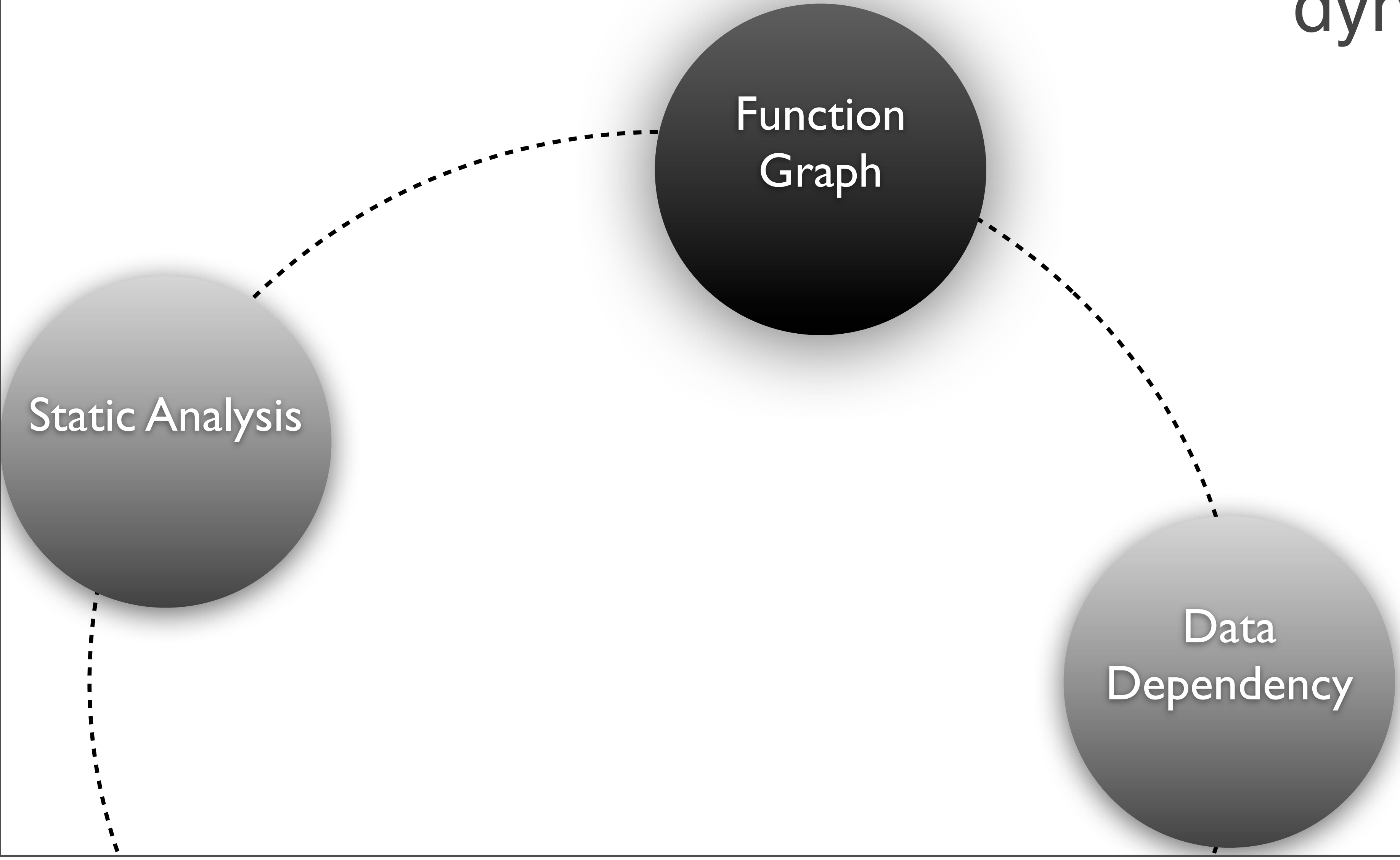
Function Graph

Static Analysis

Data Dependency

Function Graph

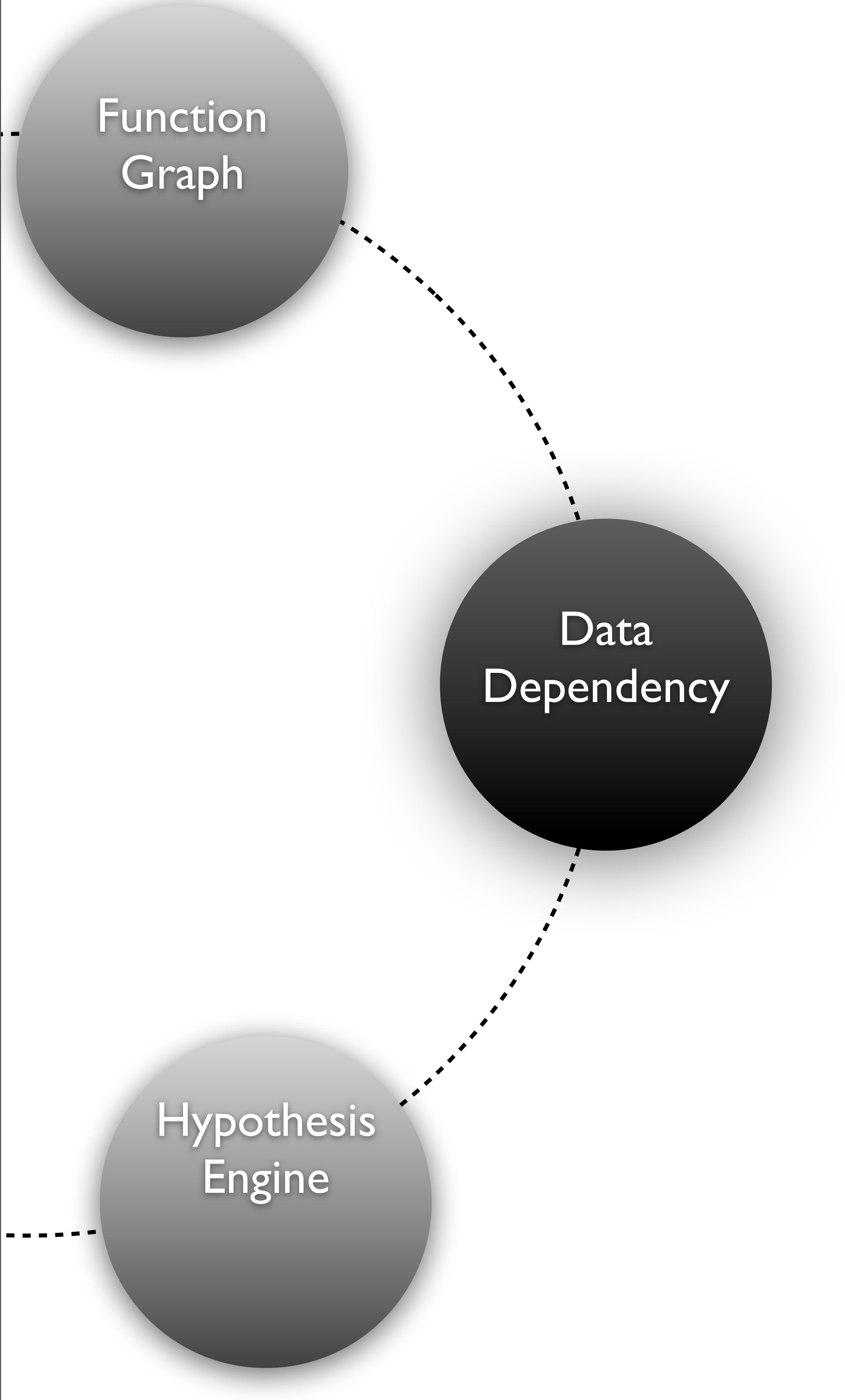
- We construct our map of the program.
- Use this for fuzzing entry points.
- “Maps” the static to dynamic

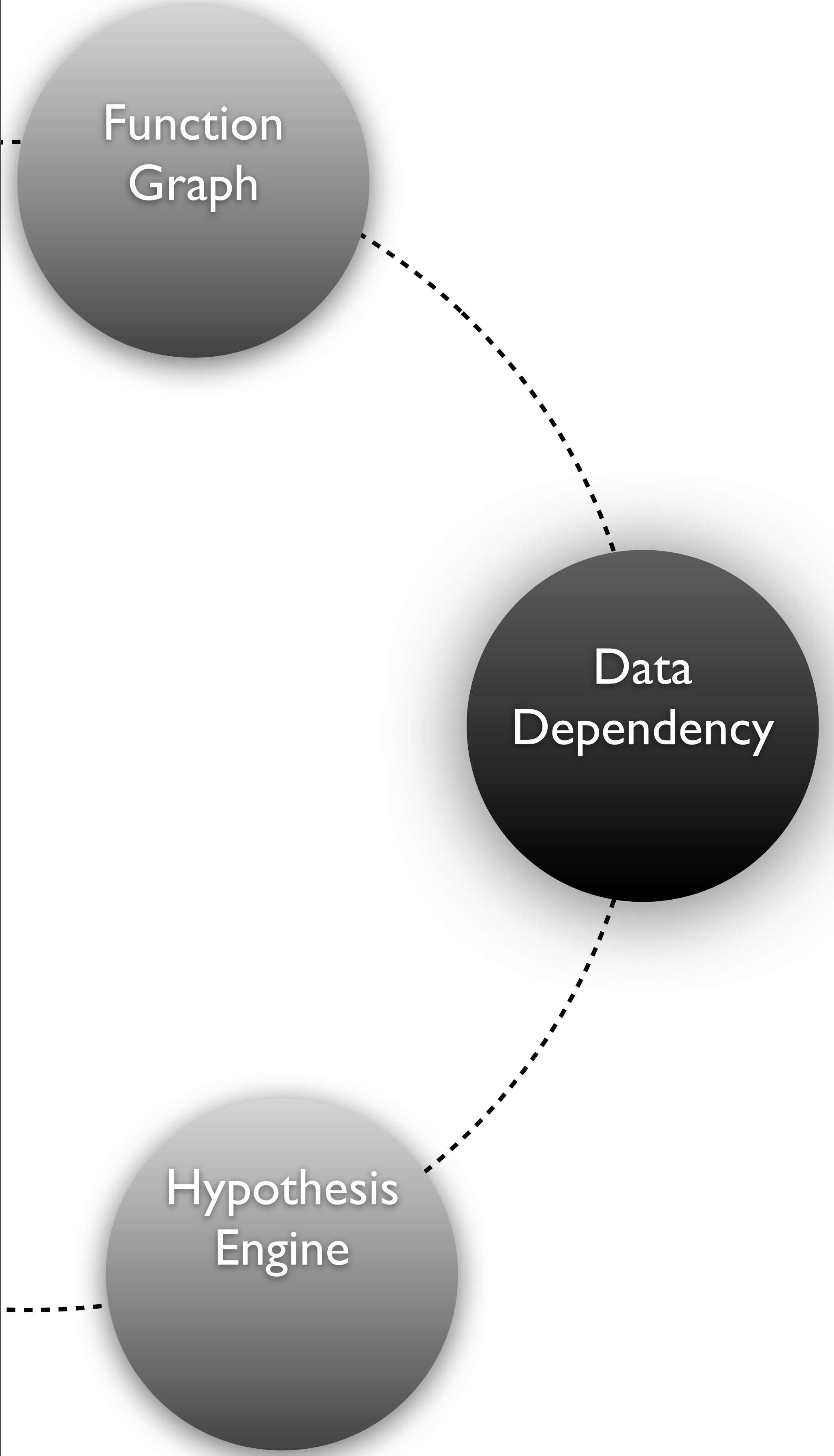


Function Graph

Static Analysis

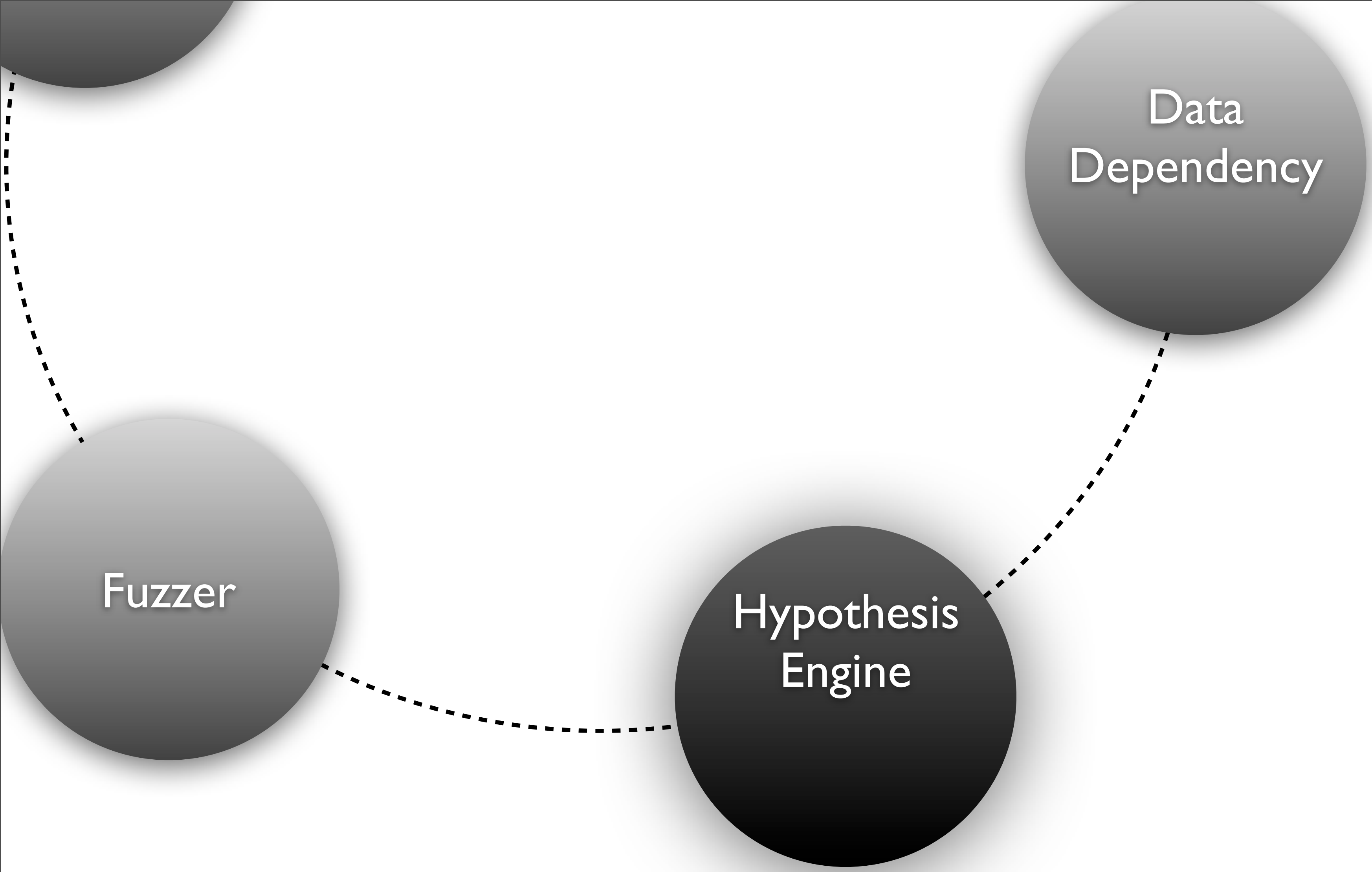
Data Dependency

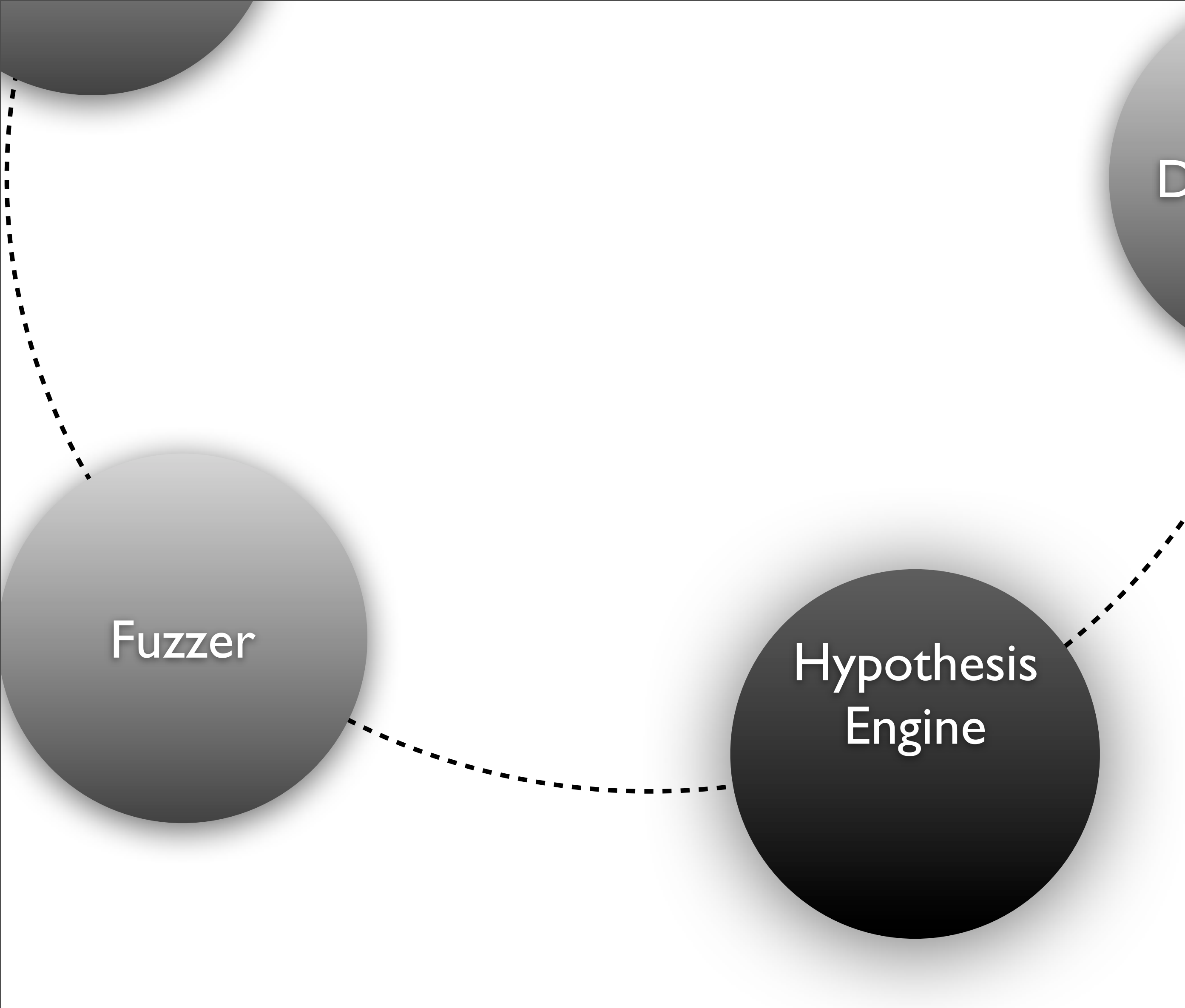

Function
Graph
Data
Dependency
Hypothesis
Engine



Data Dependencies

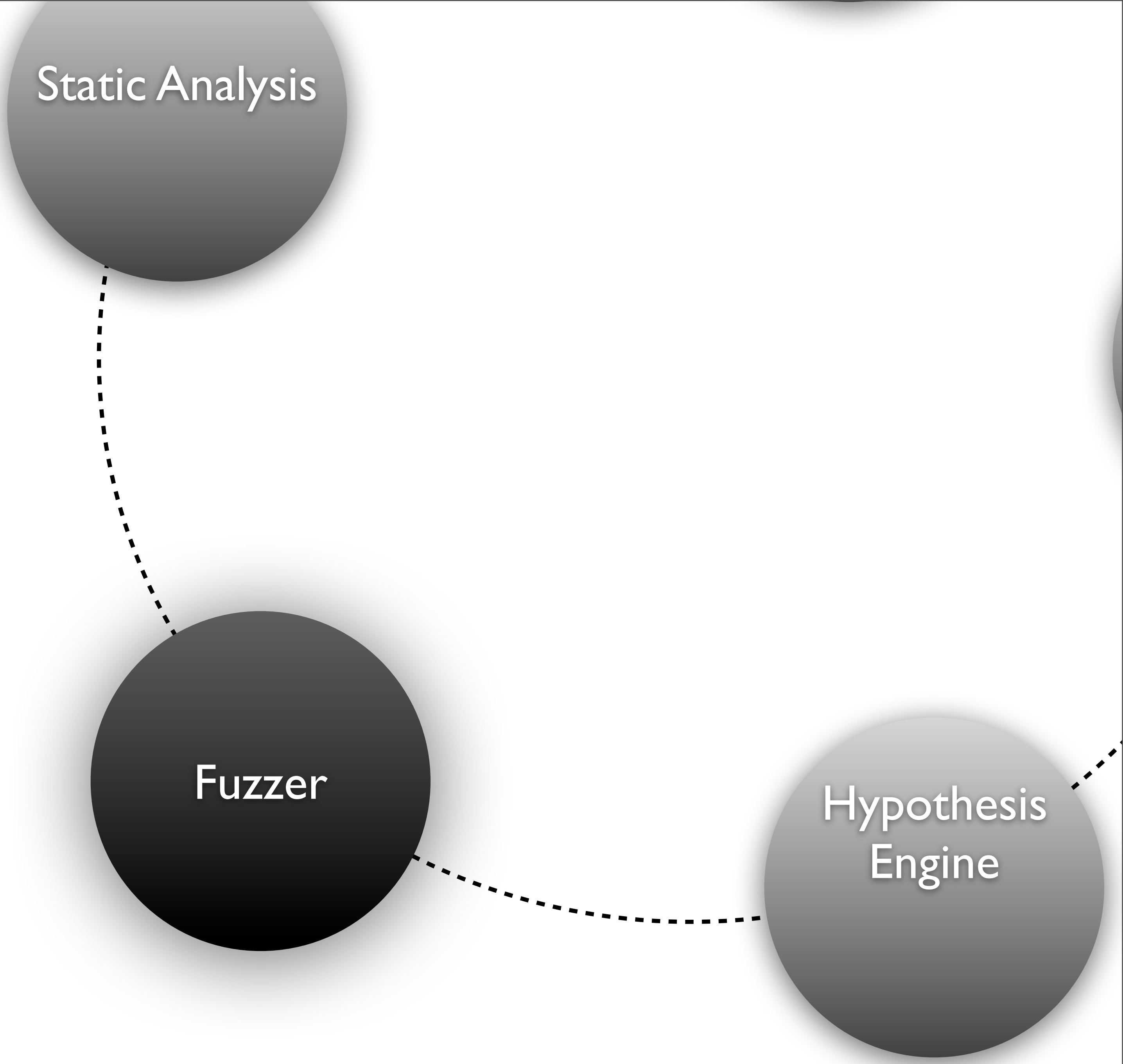
- Look at data, local variables, etc...
- Taint analysis
- Parse AST
- Store dependency relationships into "facts"


Data Dependency
Fuzzer
Hypothesis Engine



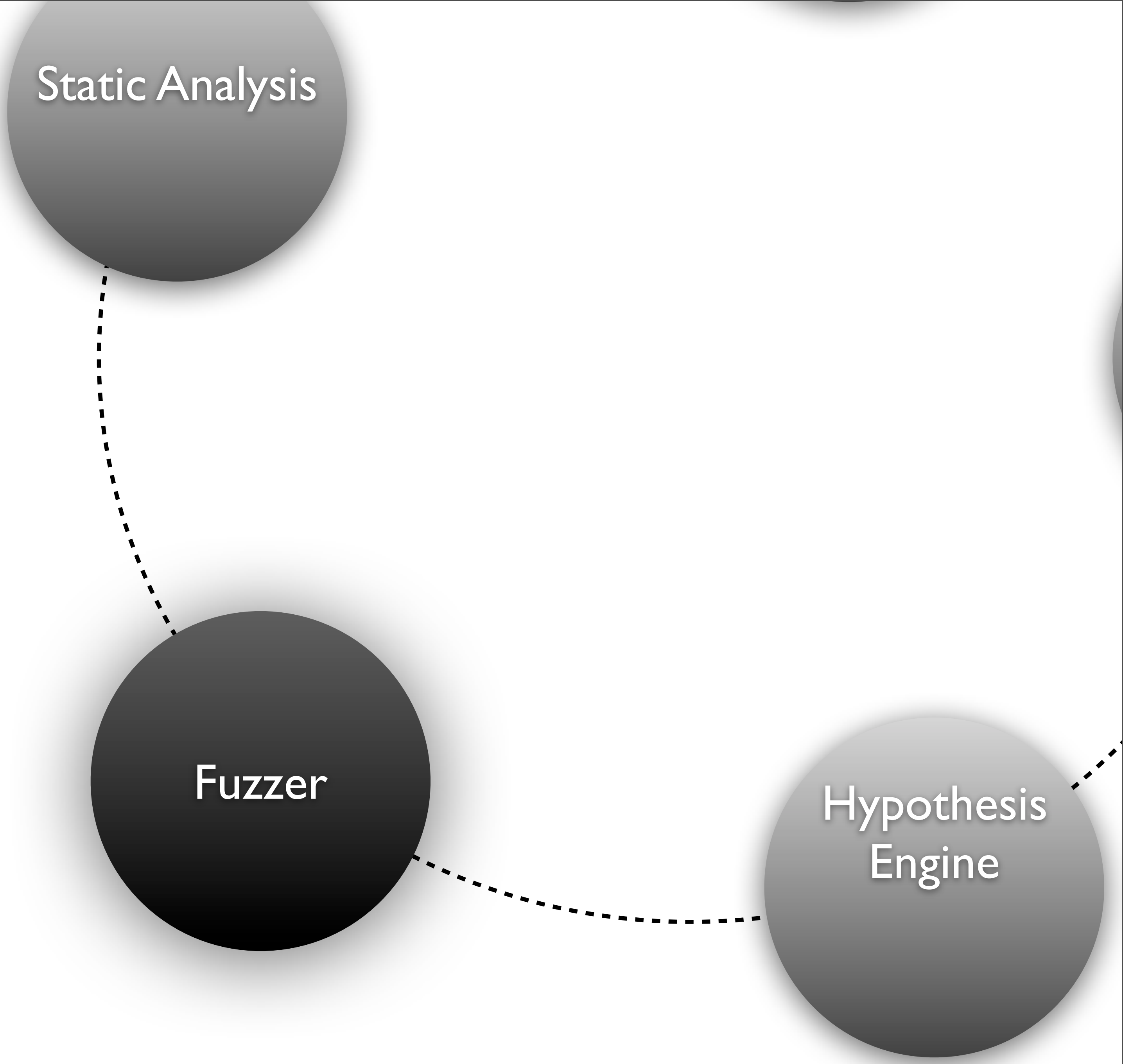
Hypothesis Engine

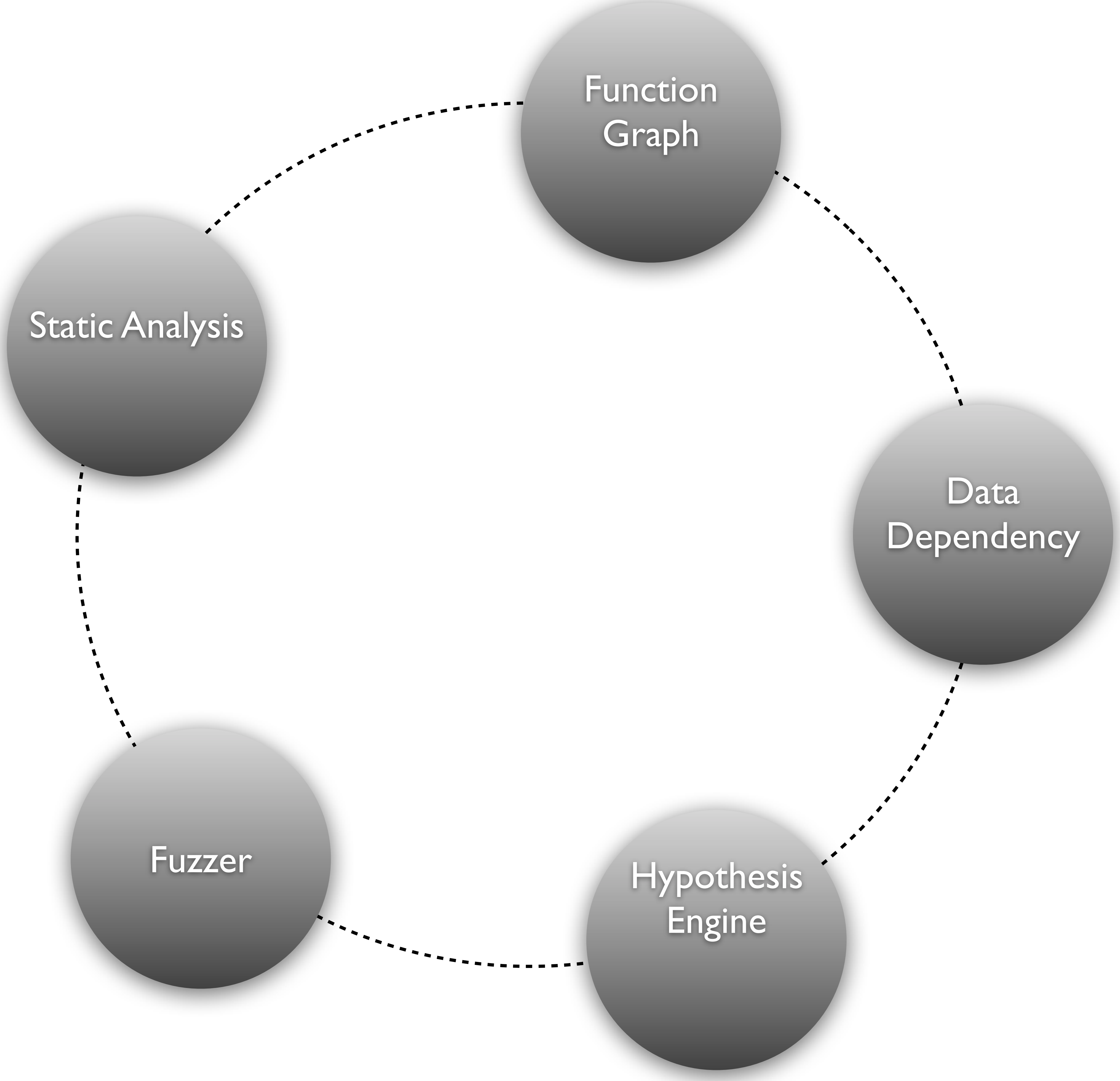
- Hypothesis Engine tracks what succeeds and what does not.
- Simple rules engine...


Static Analysis
Fuzzer
Hypothesis Engine

Fuzzer

- Utilize the LLDB python framework to drive the fuzzing
- Allows us to easily recover register, stack information, etc.
- Python allows for easy extensions...





Function Graph

Data Dependency

Hypothesis Engine

Fuzzer

Static Analysis

Static Analysis

Static Analysis

- Rewriting the scan-build/analyzer PERL scripts to Python scripts...
- Why? Masochism? Maybe a little...
- Comprehensive, this makes it easier to take a python Fuzzer and have it interact with the static analyzer.

In Depth Analysis

- With a rewrite, we can make it easier to:
  - Run non-standard checkers when needed, not just when aware
  - Import custom checkers:

```
clang++ -Xclang -load -Xclang ./MainCallChecker.so -Xclang

-analyzer-checker=example.MainCallChecker --analyze global_static.cpp
```

Function
Graph

Call Function Graph

- Anchors our static to the dynamic engine.
- We want to do this offline so we tie into hypothesis, after all this is a development tool.
- Ultimately want to present the developer with a suggestion for another approach.
    - Pie in the sky? probably, but can’t get it right unless you try...

Call Function

Mappings...

Function	Variable	Type	Input to SubFunction
_start	argv	char **	main
main	user_info	struct param*	defaults
defaults	diag_level	unsigned int	set_depth
set_depth	cur_wrk_dir	char*	readenv

Data
Dependency

Data Dependency

- Find local variables, sources, understand how these taint downstream objects.
- Leverage the CFG to follow the byte trail.
- Don’t forget, we can read/write to memory if we absolutely need to...

Local Tainting

- By understanding the CFG, we can even directly write to memory using LLDB to look at crash severity, i.e. write a test payload.
- Write “AAAA...” = 0x41414141...” or other recognizable patterns into memory to taint.

Hypothesis
Engine

Hypothesis Engine

- This is a development tool, there ought to be a “goal”!
- Let’s face it, incorrect assumptions lead to serious problems within a project.
- We can present information in this format as well as present “hypothesis” for how to break the software - understand the limits!

Facts Engine

- Using PYKE, a Python based A.I. engine that mirrors Prolog in terms of its functionality.
- Excellent fact storage.
- Draw conclusions, “goals” in PYKE Parlance, to deduce information.
- PYKE’s use of a “plan” for general or specific use cases is a valuable piece for our architecture.

Conclusions

- Draw on conclusions reached
- At a later stage, comments or other syntactic elements could be incorporated to test developer’s goals.

Fuzzer

Dynamic Instrumentation

- Simple implementation for this project is a fuzzer.
- Incorporate some aspects of Sulley to leverage LLDB rather than PyDBG
    - Record keeping, data generation most notably.

Crash Investigation

- The developer doesn't want just a log of crashes. Understanding why it crashed and severity is key!
- This means generate the crash, use LLDB to store register information, jump up a stack frame to create a hypothesis as to the crash.
- Can test against additional static analysis, as well as generating additional dynamic tests of the hypothesis.

Example of Crash

```
Register Information:
General Purpose Registers:
       rax = 0x0000000000000000
       rbx = 0x00007fff54dfcd00
       rcx = 0x00007fff54dfcce8
       rdx = 0x0000000000000000
       rdi = 0x00000000000004cf
       rsi = 0x0000000000000006
       rbp = 0x00007fff54dfcd10
       rsp = 0x00007fff54dfcce8
        r8 = 0x0000000000000000
        r9 = 0x0000000000000000
       r10 = 0x00007fff8d5a6342  libsystem_kernel.dylib`sigprocmask + 10
       r11 = 0x0000000000000206
       r12 = 0x0000000000000000
       r13 = 0x0000000000000000
       r14 = 0x0000000000000000
       r15 = 0x0000000000000000
       rip = 0x00007fff8d5a4d46  libsystem_kernel.dylib`__kill + 10
    rflags = 0x0000000000000206
        cs = 0x0000000000000007
        fs = 0x0000000000000000
        gs = 0x0000000000000000
```

LLDB python API easily grabs register state, etc

Diagnose Exploitability based on register control

Tracking Local States

- Python API; use breakpoints to “pause”
- At these junctures we can grab local state ensure we understand how the program is being traversed (similar to **dtrace** functionality).

```
while process.GetState() == lldb.eStateStopped:
    com_interpreter.HandleCommand( command, result )
    name = which_frame( result, str( target) )
    if name:
        # add our name to the iterative results
        print "We're at %s" % name
        frames.append( name )
    else:
        # let's do a register dump and kill the process
        print "Stopped process, performing register dump"
        com_interpreter.HandleCommand( "register read", result )
        fhandle.write("Execution Error:\n")
        fhandle.write("Register Information: \n%s\n" % result.GetOutput() )
        process.Destroy()
        break

    process.Continue()
```

Coupling

- Fact based storage of both source deductions and dynamic results are used.
- Next iteration uses lessons learned...
- Augment with additional checkers or even notify developer of an incomplete analysis.

Python Driver

- Alpha version delivered in September...

```
analysis_step = ”””
Analysis Step ======================================================================
Using the static–analyzer to build the products via source as well
as assemble the analysis for inputs to dynamic instrumentation phase. ”””
print colored ( analysis_step , ’yellow’ )
# run scan build ...
HtmlDir = scanbuild.Main([’clang++’,’-g’,’simple.cxx’,’-o’,’simple2’]) if HtmlDir == None :
print ”””
We didn’t produce a report, for now we flag this, but this means that ←
our
static analysis didn’t reveal anything. ”””
# −−−−−−−−−−−−−
#Step two, assemble call function graph
cfg_step = ”””
Call Function Graph ======================================================================
Building the call function graph and a few other related inputs for downstream analysis and supporting functionality .
”””
print colored ( cfg_step , ’yellow’ )
# generate callgraph and assemble
walkcallgraph . Main ( ’simple.cxx’ )
# −−−−−−−−−−−−−
#Step three , assemble for supporting Hypothesis , fact generation , plus ← report
# scanning from our analysis step hypothesis_step = ”””
Hypothesis Generation ======================================================================
Building various hypothesis , reframing meta data for more abstract representations and assembling supporting information for dynamic testing .
”””
print colored ( hypothesis_step , ’yellow’ ) # compile the facts ...
engine = knowledge_engine.engine(’’) # reveal where our issues are
scanparser . Main ( HtmlDir , ’simple.cxx’ )
# step four , run the fuzzer .. dynamic_step = ”””
Dynamic Instrumentation ======================================================================
Fuzzing the program using both predetermined pathways that analysis has come up along with more standard ( i . e . conventional ) fuzzing techniques .
”””
print colored ( dynamic_step , ’yellow’ ) #Runthefuzzer
fuzzer1 . Driver ( ’simple2’ )
# −−−−−−−−−−−−−
# Step five , summarize the results summarization_step = ”””
Summarization ======================================================================
Summarization of the analysis so far – in the Beta version of the software we allow the optional recycling of this information back to the static analysis step to allow bi–directional communication
”””
print colored ( summarization_step , ’yellow’ ) # run the summarizer
summarization . Main ( ’dynamic-instrumentation/results.txt’ )
```

ScanBuild

Mapping

Artificial Intelligence

Fuzzing

Summarization

Other plugins

- Using other modules??
- This is another reason to use Python, as there are numerous fuzzing and analysis libraries.
- Fairly straightforward for analysis. Replacing the LLDB module can be done but not a trivial operation.

Example of Incorporating Other Modules: Sulley

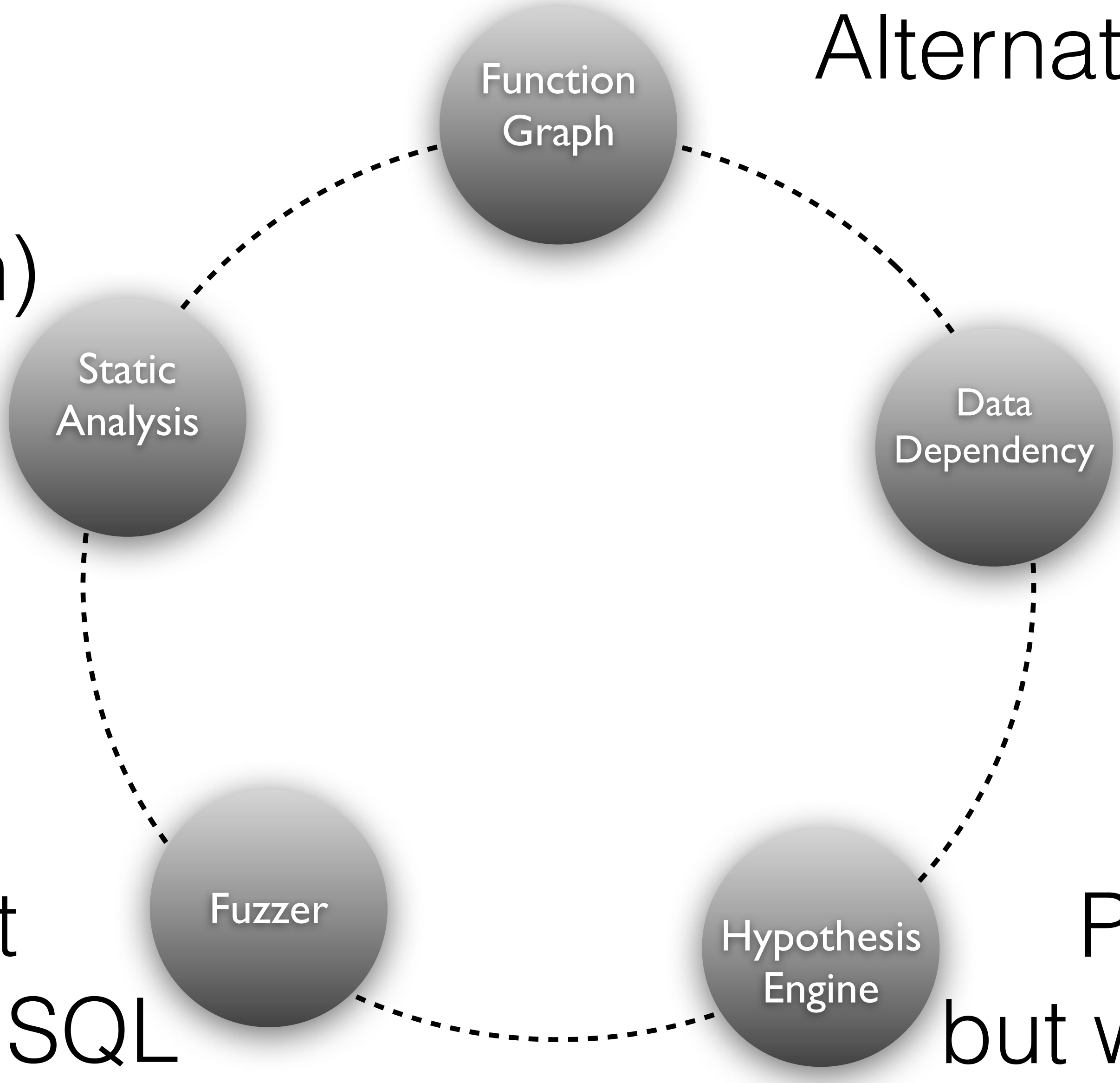
- Sulley uses **pydbg**, which is necessarily replaced by **LLDB**
- This is a substantial change within the files:

  process_monitor.py
  instrumentation.py
  pedrpc.py
- But this is only 3 of 51 files! Meaning that this is an essential but not onerous task

Important Ideas to Carry Over

- Keep it modular
- We want a general architecture that is easily customized...
- Fits well with Python modules
- Plug and play...

Envisioning a WorkFlow



Now JSON,
Alternatively SQL

Scripting
(as it as been)

Now JSON,
Alternatively SQL

Facts, but
will extend to SQL

PYKE Facts
but will translate to
SQL

Other Ideas

- By keeping this in a scripting language, we can create a distributed service without too much pain
- LLVM Interpreter to simulate for additional architectures
- Extend with additional black box techniques or modules

Project Timeline

- We’ve released an alpha to DARPA, a beta release is due towards the end of November.
- After the conclusion of this project we will contribute this to the open source community.
- Look for this in early December or thereabouts...
- Feel free to contact: jcarlson@gototheboard.com

Thank You

Thanks to....

Ayal Spitz, Alan Stone, Seth Landsman and especially Peiter Zatko, DARPA, and Bit Systems

And of course - Thanks to you for listening...

Questions?