Scalable Post-Mortem Debugging

Abel Mathew

CEO - Backtrace
amathew@backtrace.io
@nullisnt0

Debugging... or Sleeping?

Backtrace

Debugging

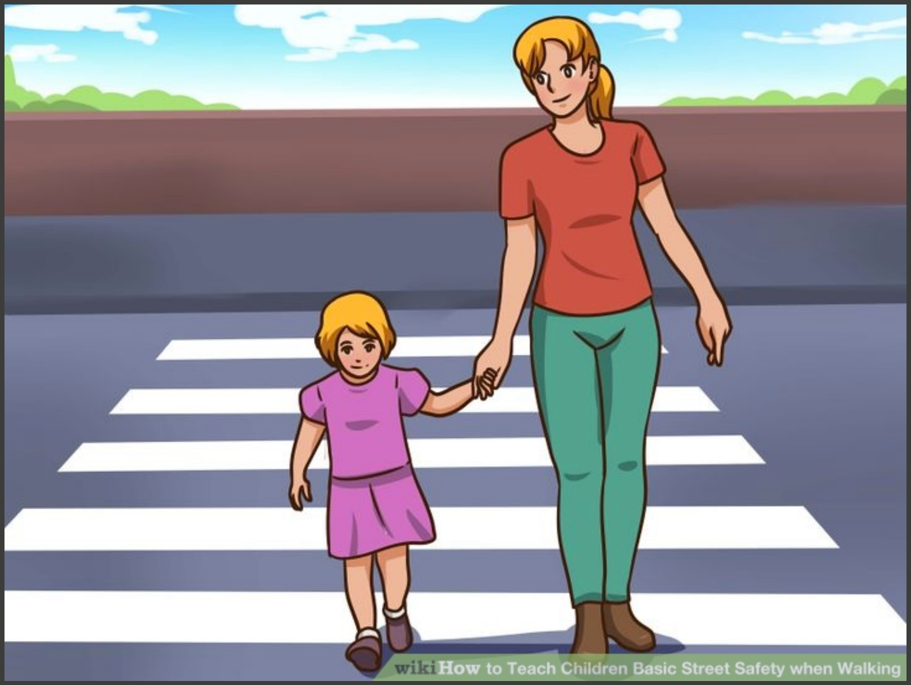
- Debugging: examining (program state, design, code, output) to identify and remove errors from software.
- Errors come in many forms: fatal, non-fatal, expected and unexpected
- The complexity of systems means more production debugging
- Pre-release tools like static analysis, model checking help catch errors before they hit production, but aren’t a complete solution.

Backtrace

Debugging Methods

- Breakpoint
- Printf/Logging/Tracing
- Post-Mortem

Breakpoint



Backtrace

Log Analysis / Tracing

- The use of instrumentation to extract data for empirical debugging.
- Useful for:
  - observing behavior between components/services (e.g. end to end latency)
  - non-fatal & transient failure that cannot otherwise be made explicit

Backtrace

Log Analysis / Tracing

- Log Analysis Systems:
  - Splunk, ELK, many others…
- Tracing Systems:
  - Dapper, HTrace, Zipkin, Stardust, X-Trace

Post-Mortem Debugging

- Using captured program state from a point-in-time to debug failure post-mortem or after-the-fact
- Work back from invalid state to make observations about how the system got there.
- Benefits:
  - No overhead except for when state is being captured (at the time of death, assertion, explicit failure)
  - Allows for a much richer data set to be captured
  - Investigation + Analysis is done independent of the failing system’s lifetime.
  - Richer data + Independent Analysis == powerful investigation

Backtrace

Post-Mortem Debugging

- Rich data set also allows you to make observations about your software beyond fixing the immediate problem.
- Real world examples include:
  - leak investigation
  - malware detection
  - assumption violation

Post-Mortem Facilities

- Most operating environments have facilities in place to extract dumps from a process.
- How do you get this state?
- How do you interpret it?

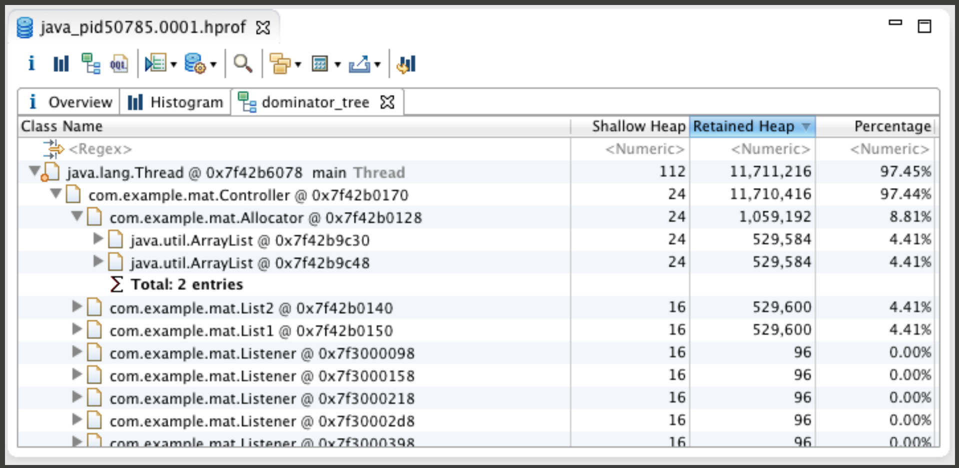
PMF: Java

- Extraction: heap dumps
  - `-XX:+HeapDumpOnOutOfMemoryError`
  - Can use `jmap -dump:[live,]format=b,file=<filename> <PID>` on a live process or core dump
    - Can filter out objects based on “liveness”
    - Note: this will pause the JVM when running on a live process
- Extraction: stack traces / “thread dump”
  - Send `SIGQUIT` on a live process
  - `jstack <process | core dump>`
    - `-l` prints out useful lock and synchronization information
    - `-m` prints both Java and native C/C++ frames

Backtrace

PMF: Java

- Inspecting heap dumps: Eclipse MAT
- Visibility into shallow heap, retained heap, dominator tree.

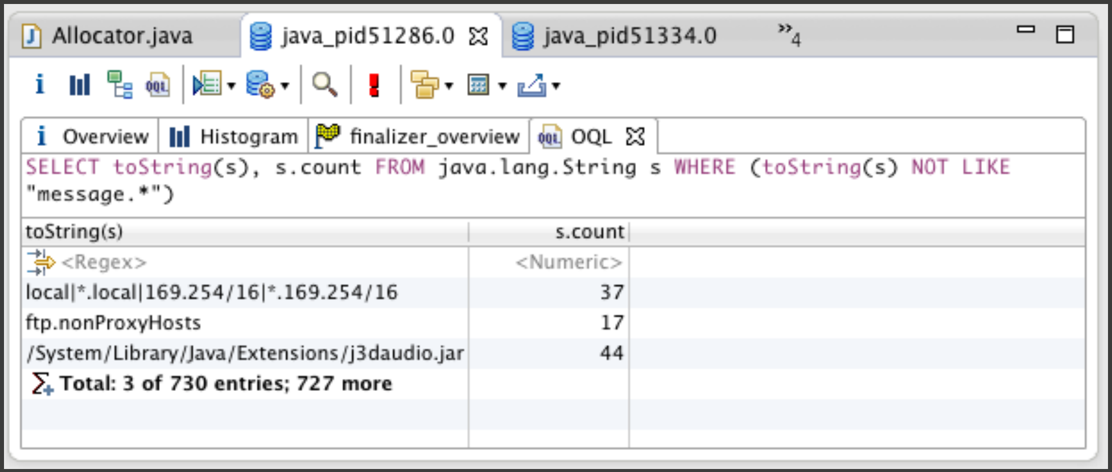


Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.lang.Thread @ 0x7f42b6078 main Thread	112	11,711,216	97.45%
com.example.mat.Controller @ 0x7f42b0170	24	11,710,416	97.44%
com.example.mat.Allocator @ 0x7f42b0128	24	1,059,192	8.81%
java.util.ArrayList @ 0x7f42b9c30	24	529,584	4.41%
java.util.ArrayList @ 0x7f42b9c48	24	529,584	4.41%
∑ **Total: 2 entries**			
com.example.mat.List2 @ 0x7f42b0140	16	529,600	4.41%
com.example.mat.List1 @ 0x7f42b0150	16	529,600	4.41%
com.example.mat.Listener @ 0x7f3000098	16	96	0.00%
com.example.mat.Listener @ 0x7f3000158	16	96	0.00%
com.example.mat.Listener @ 0x7f3000218	16	96	0.00%
com.example.mat.Listener @ 0x7f30002d8	16	96	0.00%
com.example.mat.Listener @ 0x7f3000398	16	96	0.00%

http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/

PMF: Java

- Inspecting heap dumps: jhat
- Both MAT and jhat expose OQL to query heap dumps for, amongst other things, differential analysis.



```
SELECT toString(s), s.count FROM java.lang.String s WHERE (toString(s) NOT LIKE
"message.*")
```

toString(s)	s.count
<Regex>	<Numeric>
local|*.local|169.254/16|*.169.254/16	37
ftp.nonProxyHosts	17
/System/Library/Java/Extensions/j3daudio.jar	44
Total: 3 of 730 entries; 727 more	

http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/

Backtrace

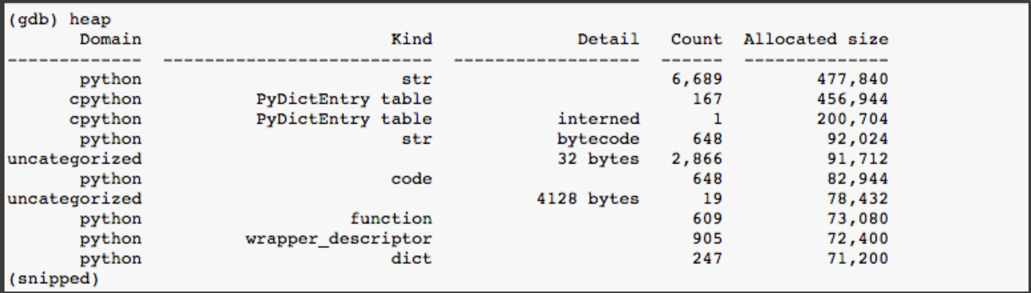
PMF: Python

- Extraction: `os.abort()` or running `` `gcore` `` on the process
- Inspection: gdbinit — a number of macros to interpret Python cores
  - `py-list`: lists python source code from frame context
  - `py-bt`: Python level backtrace
  - `pystackv`: get a list of Python locals with each stack frame

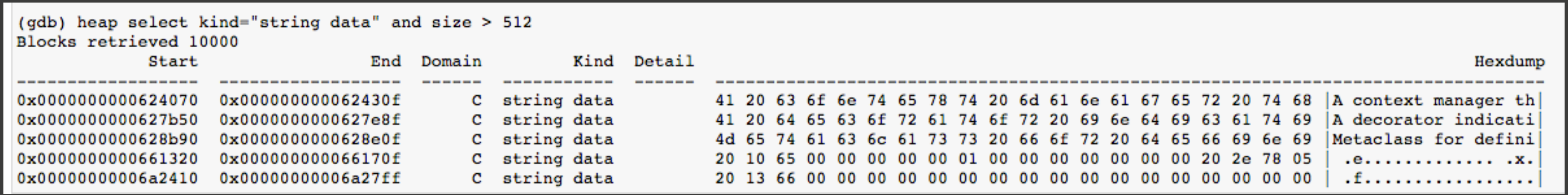
Backtrace

PMF: Python

- gdb-heap — extract statistics on object counts, etc. Provides “heap select” to query the Python heap.



```
(gdb) heap
       Domain                        Kind              Detail   Count  Allocated size
-------------  --------------------------  ------------------  ------  --------------
       python                         str                       6,689         477,840
      cpython           PyDictEntry table                         167         456,944
      cpython           PyDictEntry table            interned       1         200,704
       python                         str            bytecode     648          92,024
uncategorized                                        32 bytes   2,866          91,712
       python                        code                         648          82,944
uncategorized                                      4128 bytes      19          78,432
       python                    function                         609          73,080
       python          wrapper_descriptor                         905          72,400
       python                        dict                         247          71,200
(snipped)
```



```
(gdb) heap select kind="string data" and size > 512
Blocks retrieved 10000
             Start                 End  Domain         Kind  Detail                                                                              Hexdump
------------------  ------------------  ------  -----------  ------  ----------------------------------------------------------------------------------
0x0000000000624070  0x000000000062430f       C  string data          41 20 63 6f 6e 74 65 78 74 20 6d 61 6e 61 67 65 72 20 74 68 |A context manager th|
0x0000000000627b50  0x0000000000627e8f       C  string data          41 20 64 65 63 6f 72 61 74 6f 72 20 69 6e 64 69 63 61 74 69 |A decorator indicati|
0x0000000000628b90  0x0000000000628e0f       C  string data          4d 65 74 61 63 6c 61 73 73 20 66 6f 72 20 64 65 66 69 6e 69 |Metaclass for defini|
0x0000000000661320  0x000000000066170f       C  string data          20 10 65 00 00 00 00 00 01 00 00 00 00 00 00 00 20 2e 78 05 | .e............ .x.|
0x00000000006a2410  0x00000000006a27ff       C  string data          20 13 66 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .f.................|
```

PMF: Go

- Basic tooling available via lldb & mdb.
- GOTRACEBACK=crash environment variable enables core dumps

PMF: Node.js

- `—abort_on_uncaught_exception` generates a coredump
- Rich tooling for mdb and llnode to provide visibility into the heap, object references, stack traces and variable values from a coredump
- Commands:
  - `jsframe -iv`: shows you frames with parameters
  - `jsprint`: extracts variable values
  - `findjsobjects`: find reference object type and their children

Backtrace

PMF: Node.js

- *Debugging Node.js in Production* @ Netflix by Yunong Xiao goes in-depth on solving a problem in Node.JS using post-mortem analysis
  - Generates coredumps on Netflix Node.JS processes to investigate memory leak
  - Used `findjsobject` to find growing object counts between coredumps
  - Combining this with `jsprint` and `findjsobject -r` to find that for each `` `require` `` that threw an exception, module metadata objects were "leaked"

PMF: C/C++

- The languages we typically associate post-mortem debugging with.
- Use standard tools like gdb, lldb to extract and analyze data from core dumps.
- Commercial and open-source (core-analyzer) tools available to automatically highlight heap mismanagement, pointer corruption, function constraint violations, and more

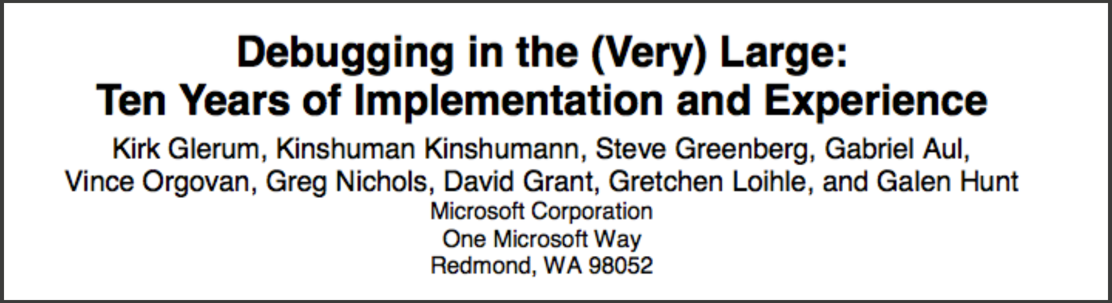
Backtrace

Scalable?

- With massive, distributed systems, one off investigations are no longer feasible.
- We can build systems that automate and enhance post-mortem analysis across components and instances of failure.
- Generate new data points that come from "debugging failure at large."
- Leverage the rich data set to make deeper observations about our software, detect latent bugs and ultimately make our systems more reliable.

Backtrace

Microsoft's WER


Debugging in the (Very) Large:
Ten Years of Implementation and Experience

Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul,
Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

- Microsoft's distributed post-mortem debugging system used for Windows, Office, internal systems and many third-party vendors.

- In 2009: "*WER is the largest automated error-reporting system in existence. Approximately one billion computers run WER client code*"

WER

- "*WER collects error reports for crashes, non-fatal assertion failures, hangs, setup failures, abnormal executions, and device failures.*"
- Automated the collection of memory dumps, environmental data, configuration, etc
- Automated the diagnosis, and in some cases, the resolution of failure
- … with very little human effort

Backtrace

WER

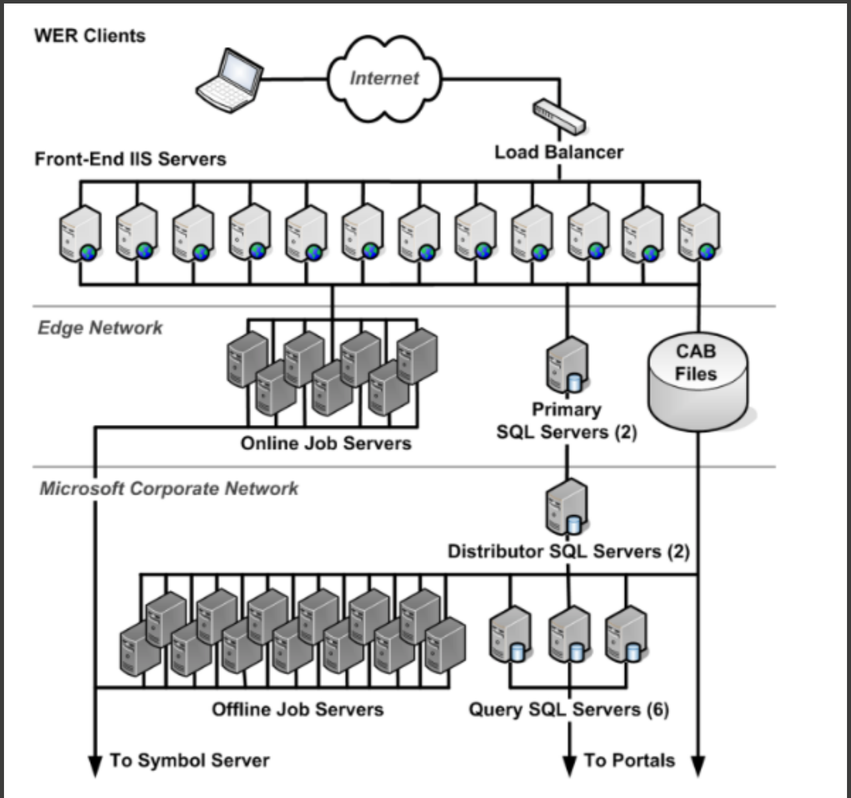


Figure 8. WER Servers.

WER: Automation

- *"For example, in February 2007, users of Windows Vista were attacked by the Renos malware. If installed on a client, Renos caused the Windows GUI shell, explorer.exe, to crash when it tried to draw the desktop. The user's experience of a Renos infection was a continuous loop in which the shell started, crashed, and restarted. While a Renos-infected system was useless to a user, the system booted far enough to allow reporting the error to WER—on computers where automatic error reporting was enabled—and to receive updates from WU."*

Backtrace

WER: Automation

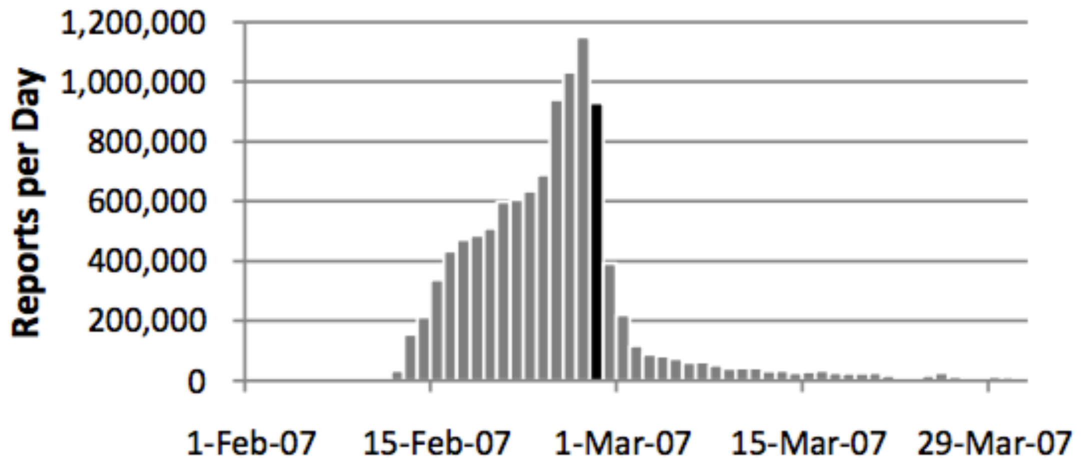


Figure 10. Renos Malware: Number of error reports per day. Black bar shows when the fix was released through WU.

WER: Bucketing

- WER aggregated errors from items through *labels* and *classifiers*
- ***labels:* use client-side info to key error reports on the "same bug"**
  - program name, assert & exception code
- ***classifiers:* insights meant to maximize programmer effectiveness**
  - heap corruption, image/program corruption, malware identified
- Bucketing extracts failure volumes by type, which helped with prioritization
- Buckets enabled automatic failure type detection which allowed automated failure response.

Backtrace

WER

Basic grouping/bucketing

Deeper analysis (!analyze)

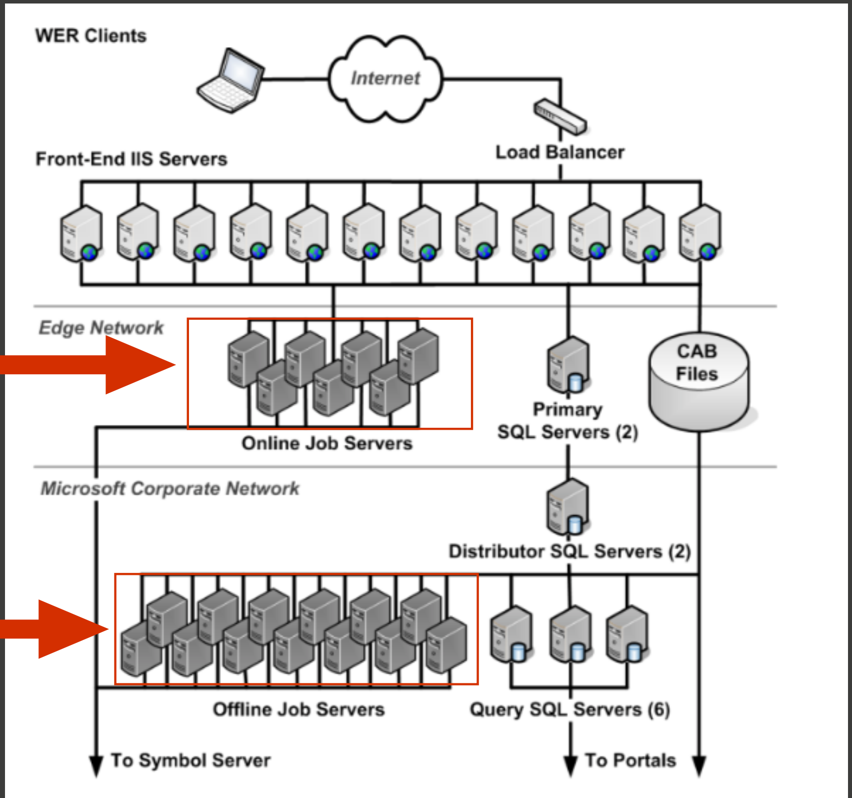


Figure 8. WER Servers.

Backtrace

WER: SBD

Statistics-based debugging

- With a rich data set, WER enabled developers to find correlations with invalid program state and outside characteristics.
- “stack sampling” helped them pinpoint frequently occurring functions in faults (instability or API misuse)
- Programmers could evaluate hypotheses on component behavior against large sets of memory dumps

Backtrace

Post-Mortem Analysis

- Only incurs overhead at the time of failure
- Allows for a more rich data set, in some cases the complete program state, to be captured
- The system can be restarted independent of analysis of program state which enables deep investigation.

Backtrace

Scalable Post-Mortem Analysis

- Scalable Post-Mortem Analysis
  - “Debugging at Large”
    - Multiple samples to test hypothesis against
    - Correlate failure with richer set of variables
- Automate detection, response, triage, and resolution of failures

Backtrace

Scalable Post-Mortem Debugging

Abel Mathew

CEO - Backtrace
amathew@backtrace.io
@nullisnt0

Backtrace