Statically-informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities

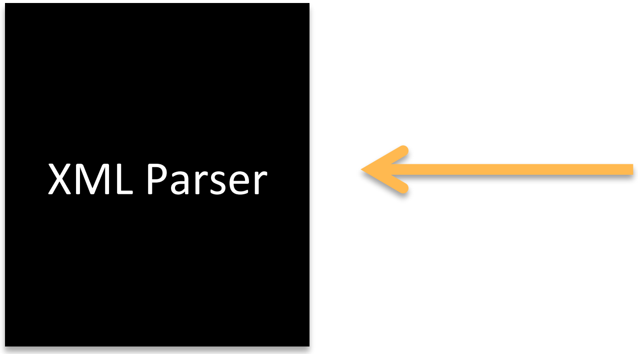
16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2016)
October 2, 2016

Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari
Email: {bholland, gsanthan, payas, kothari}@iastate.edu

Acknowledgement: Team members at Iowa State University and EnSoft, DARPA contracts FA8750-12-2-0126 & FA8750-15-2-0080

Motivation

- DARPA Space/Time Analysis for Cybersecurity (STAC) program
  - Given a compiled Java bytecode program
  - Discover *Algorithmic Complexity* (AC) vulnerabilities




```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
…
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Parsing a specially crafted input file of less than a kilobyte creates a string of 10^9 concatenated “lol” strings requiring approximately 3 gigabytes of memory.

Motivation

- DARPA Space/Time Analysis for Cybersecurity (STAC) program
  - Given a compiled Java bytecode program
  - Discover *Algorithmic Complexity* (AC) vulnerabilities
  - Vulnerabilities are defined with respect to a budget
    - Example: Max input size 1kb, execution time exceeds 300s on a given reference platform

Overview

- Approach
- Static and Dynamic Analysis Tools
  - Static loop analysis
  - Instrumentation and dynamic analysis
- Case Study
  - Walkthrough analysis
- Q/A

Approach

- Algorithmic complexity (AC) vulnerabilities are rooted in the space and time complexities of externally-controlled execution paths with loops.
  - Existing tools for computing the loop complexity are limited and cannot prove termination for several classes of loops.
  - At the extreme, a completely automated detection of AC vulnerabilities amounts to solving the intractable halting problem.
- Key Idea: Combine human intelligence with static and dynamic analysis to achieve scalability and accuracy.
  - A lightweight static analysis is used for a scalable exploration of loops in bytecode from large software, and an analyst selects a small subset of these loops for further evaluation using a dynamic analysis for accuracy.

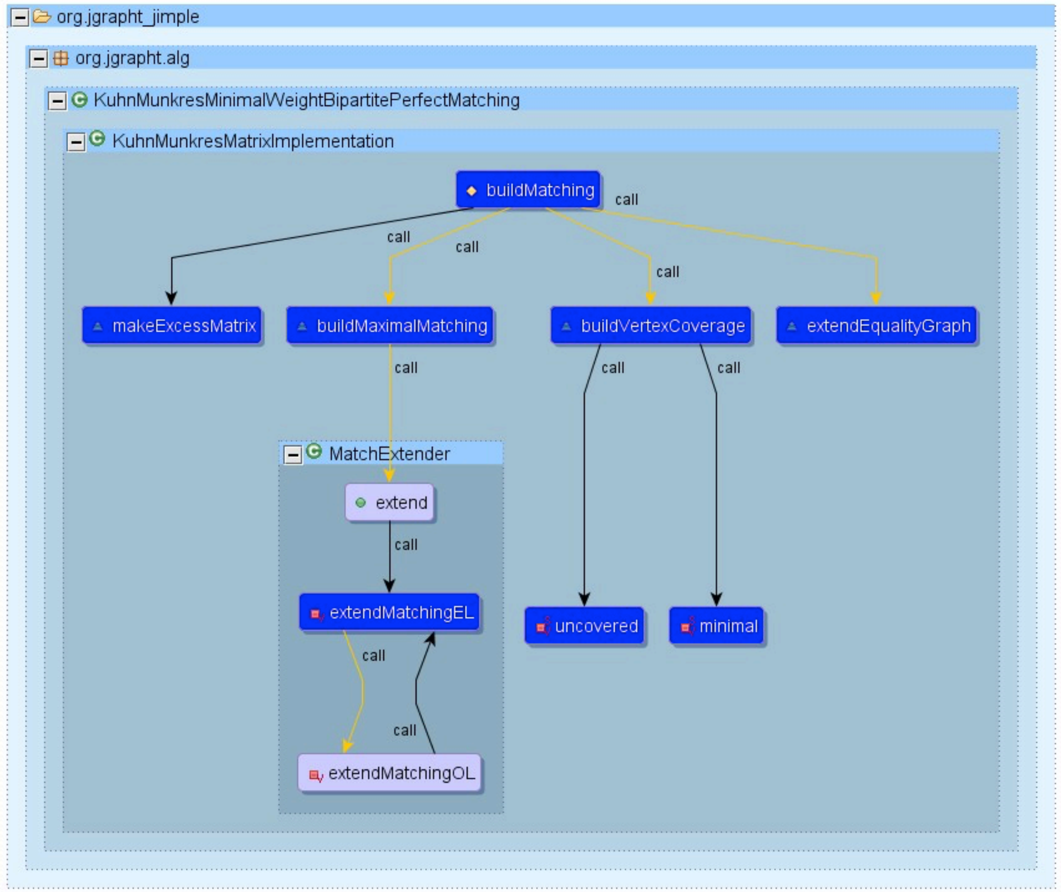
Vulnerability Detection Process

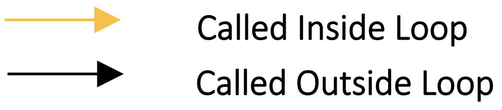
1. *Automated Exploration:* Identify loops, pre-compute their crucial attributes such as intra- and inter-procedural nesting structures and depths, and termination conditions.
2. *Hypothesis Generation:* Through an interactive inspection of the pre-computed information the analyst hypothesizes plausible AC vulnerabilities and selects candidate loops for further examination using dynamic analysis.
3. *Hypothesis Validation:* The analyst inserts probes and creates a driver to exercise the program by feeding workloads to measure resource consumption for the selected loops.

Statically-informed Dynamic Analysis (SID) Tools

- Loop Call Graph (LCG)
  - Recovers loop headers in bytecode using the DLI algorithm [Wei SAS 2007]
  - Combines call relationships to produce a compact visual model to explore intra- and inter-procedural nesting structures of loops.
  - Constructed statically, interactive, expandable, corresponds to source
- Time Complexity Analyzer (TCA)
  - A dynamic analyzer that enables the analyst to automatically instrument the selected loops with resource usage probes
  - Skeleton driver generation
  - Linear regression to estimate complexity

Loop Call Graph





Nodes:

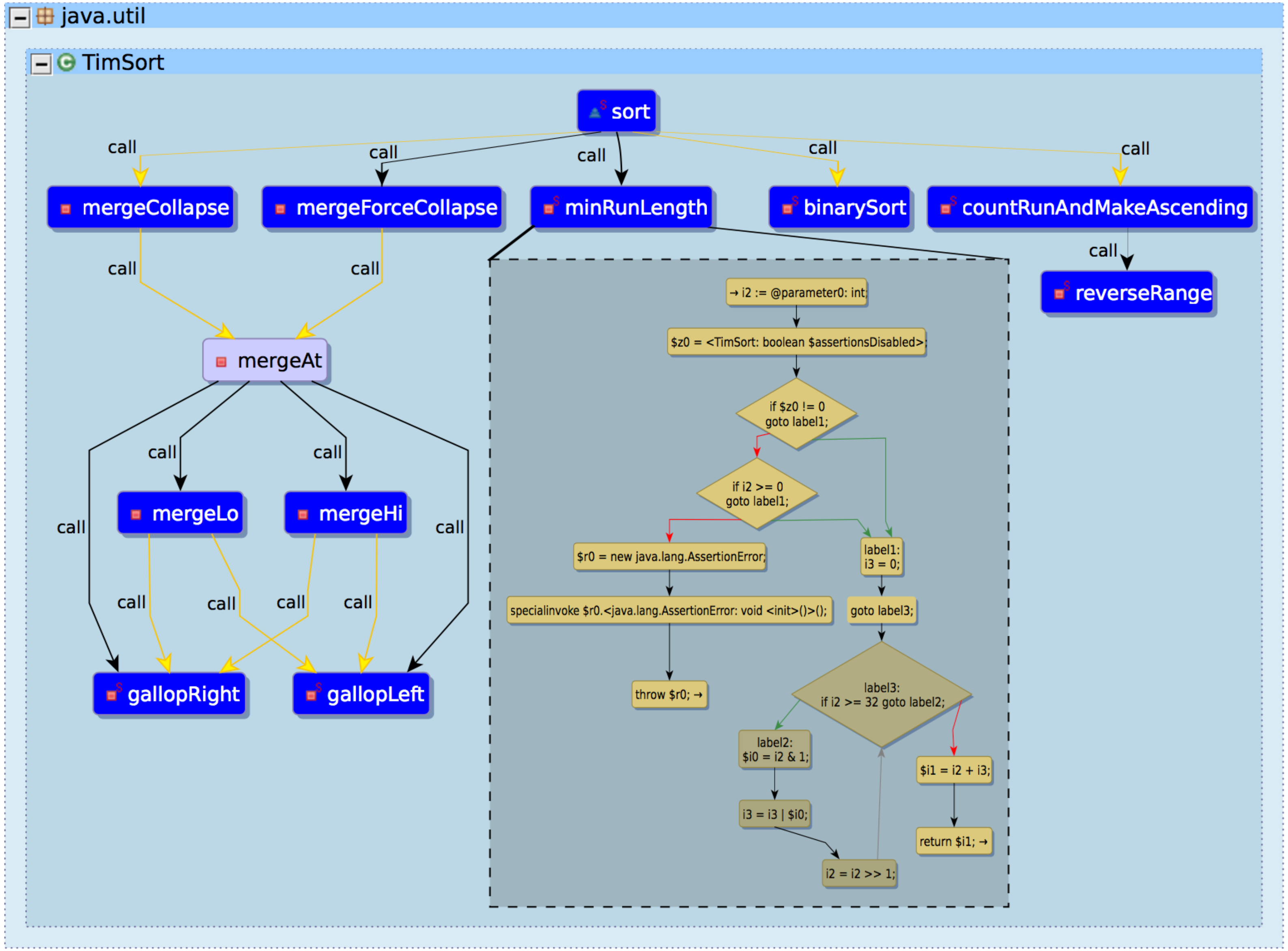
- Methods containing loops (blue)
- Methods reaching methods containing loops (white)

Edges:

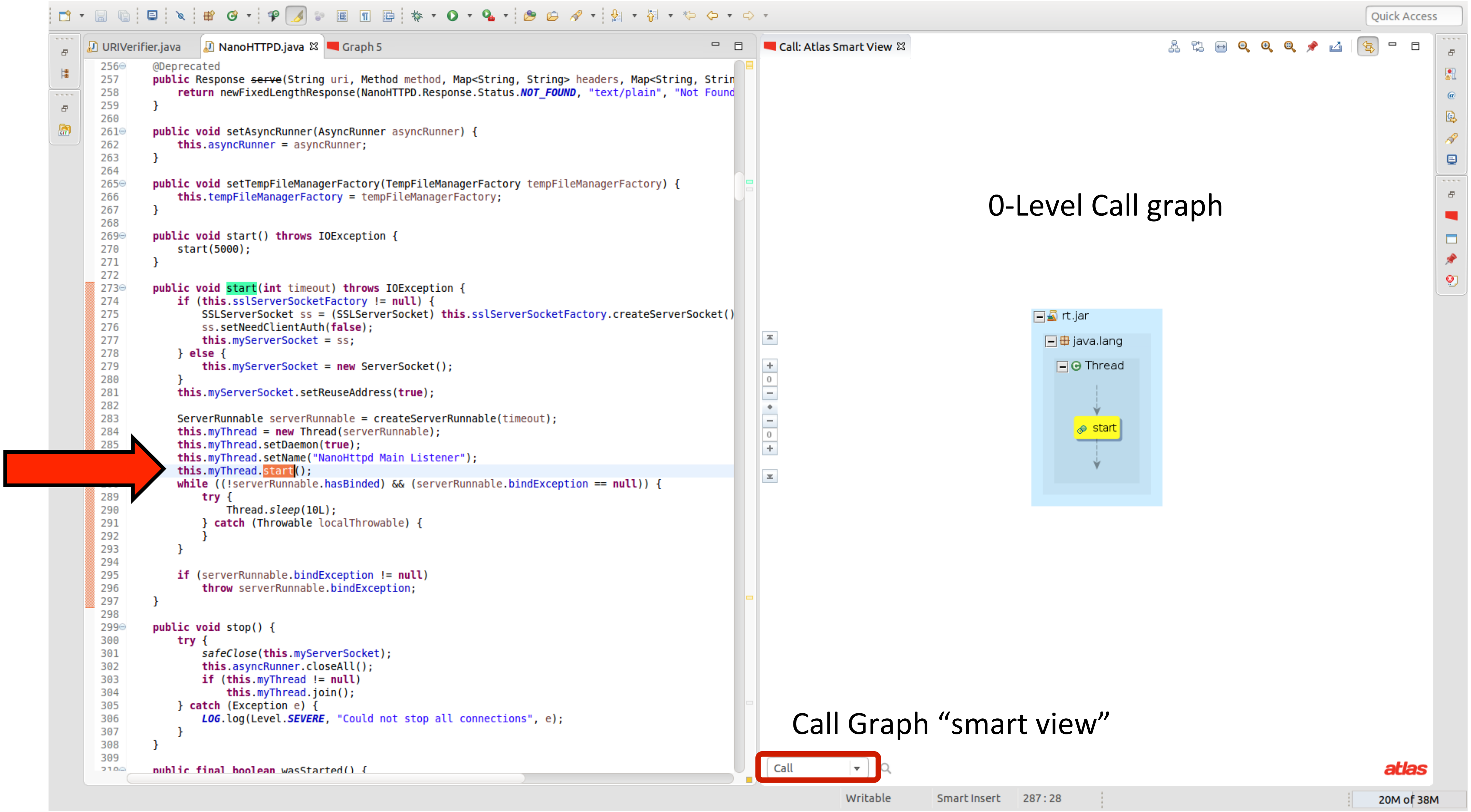
- Call relationships
- Color attributes to show placement of call site in loop

Control Flow Loop View

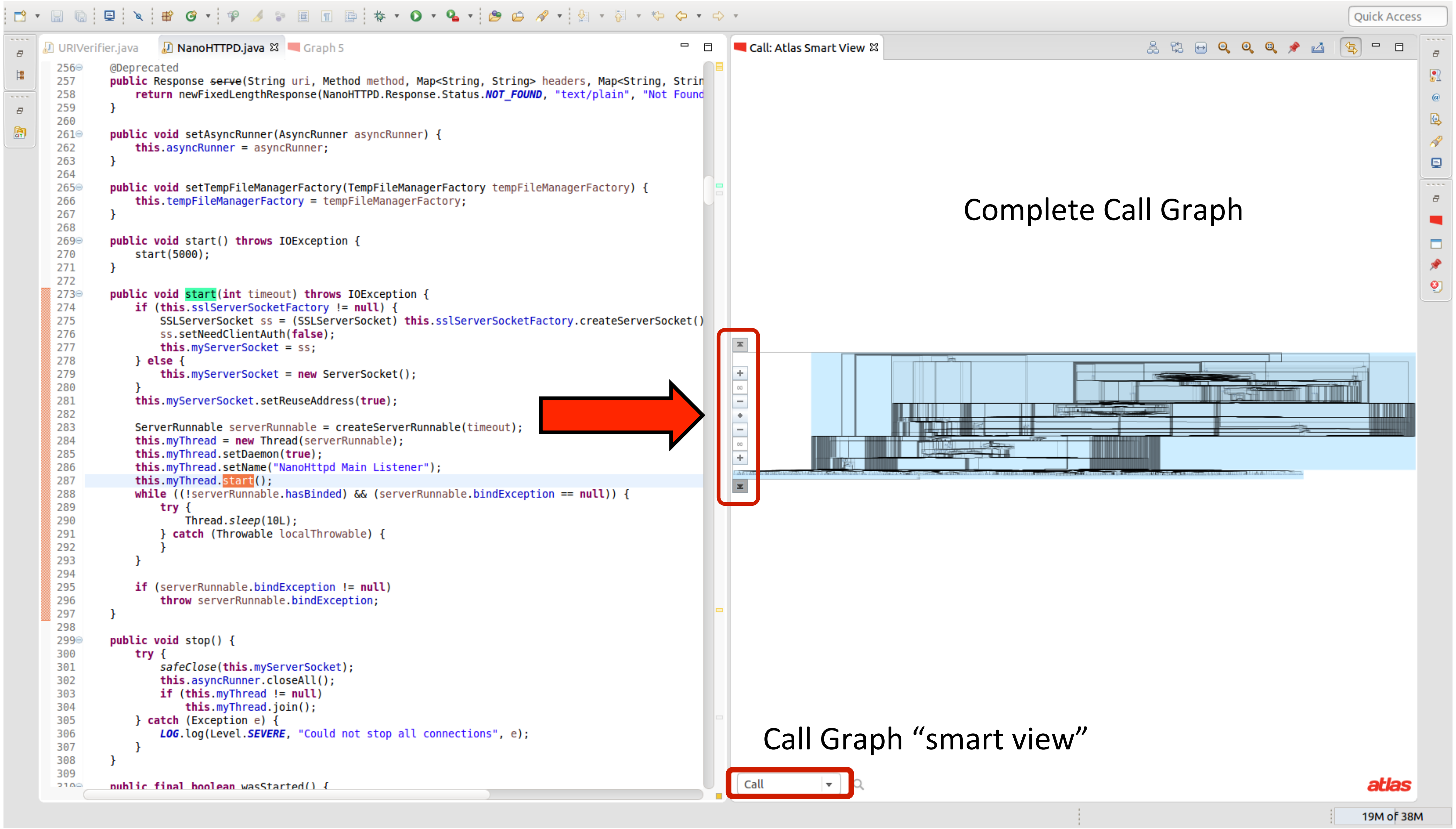
- Loop levels are shaded darker for each nesting level
- Branch condition coloring
  - Red is false
  - Green is true
- Loop back edge is grey
- Unconditional is black



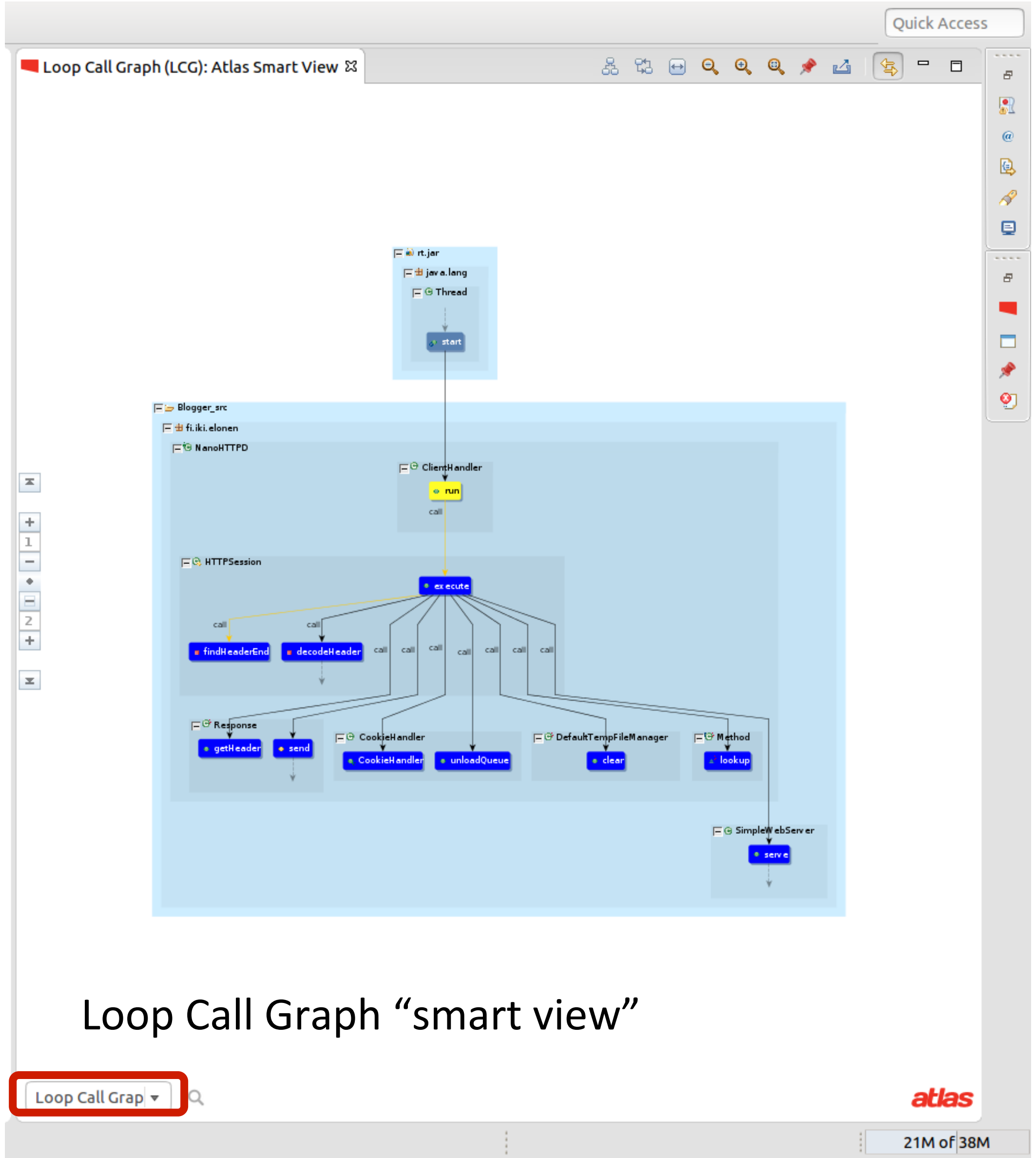
Interactive Graph Models – Traditional Call Graph



Interactive Graph Models – Traditional Call Graph

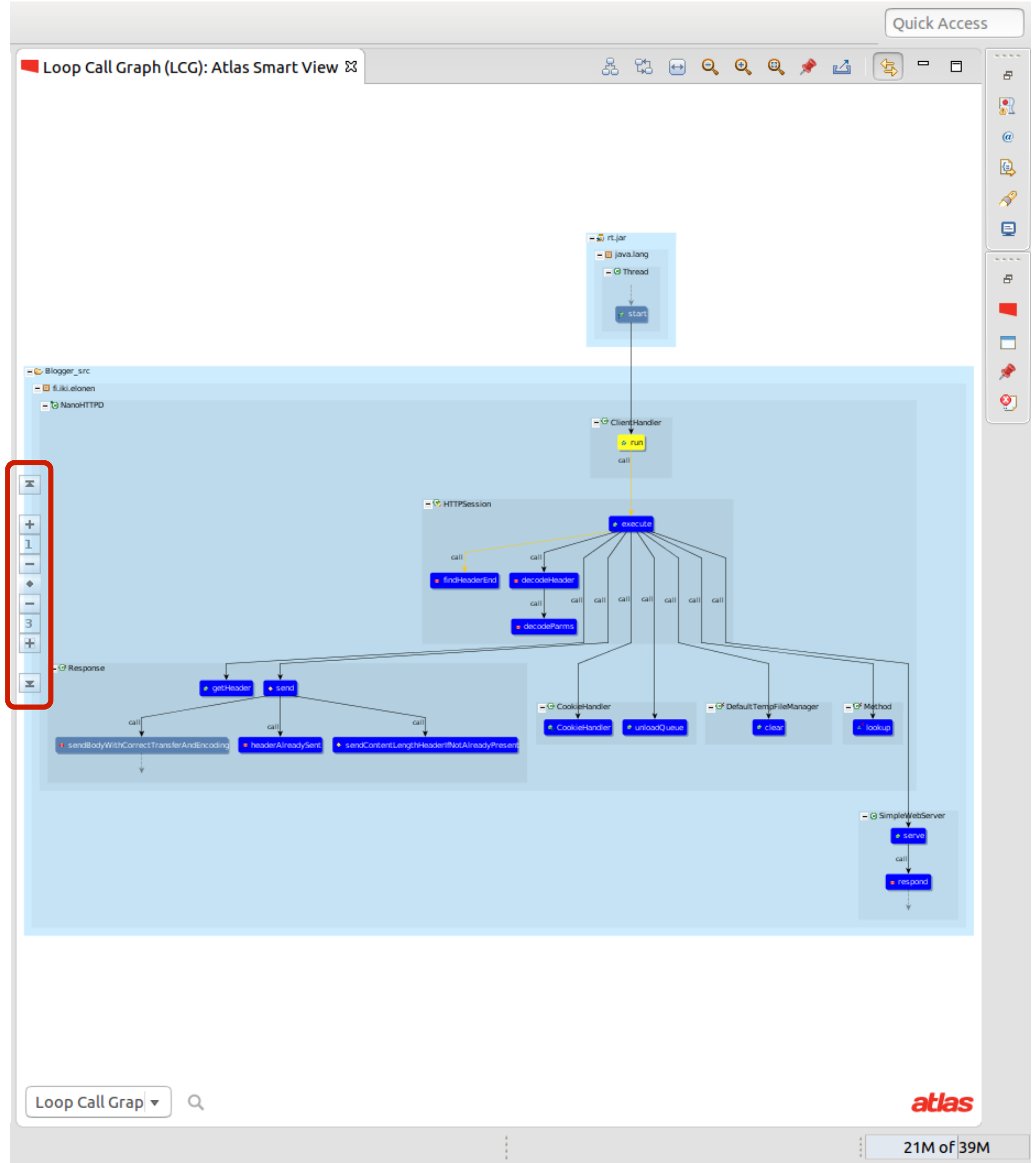


Interactive Graph Models – Loop Call Graph (Expandable)

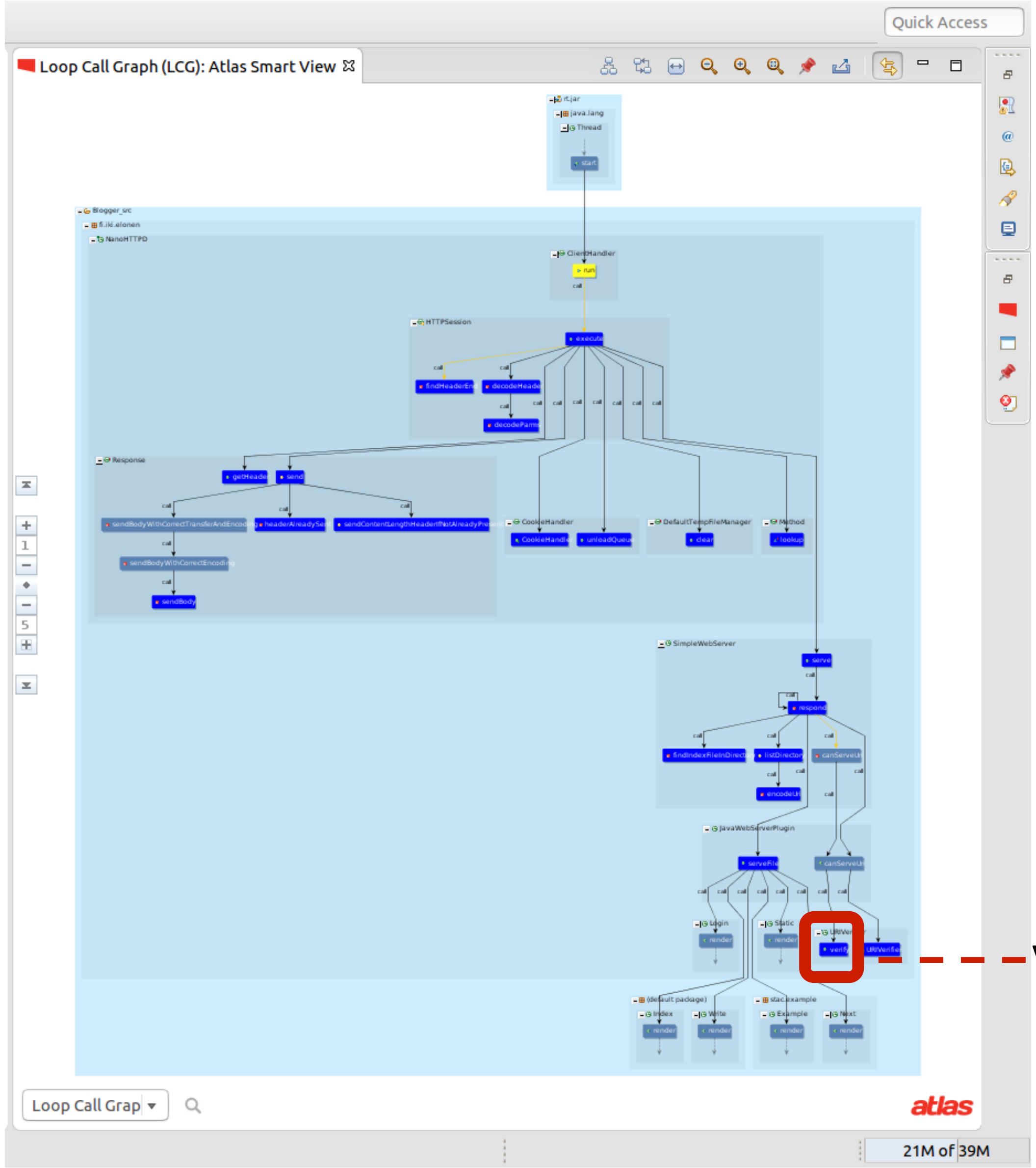


Loop Call Graph “smart view”

expandable



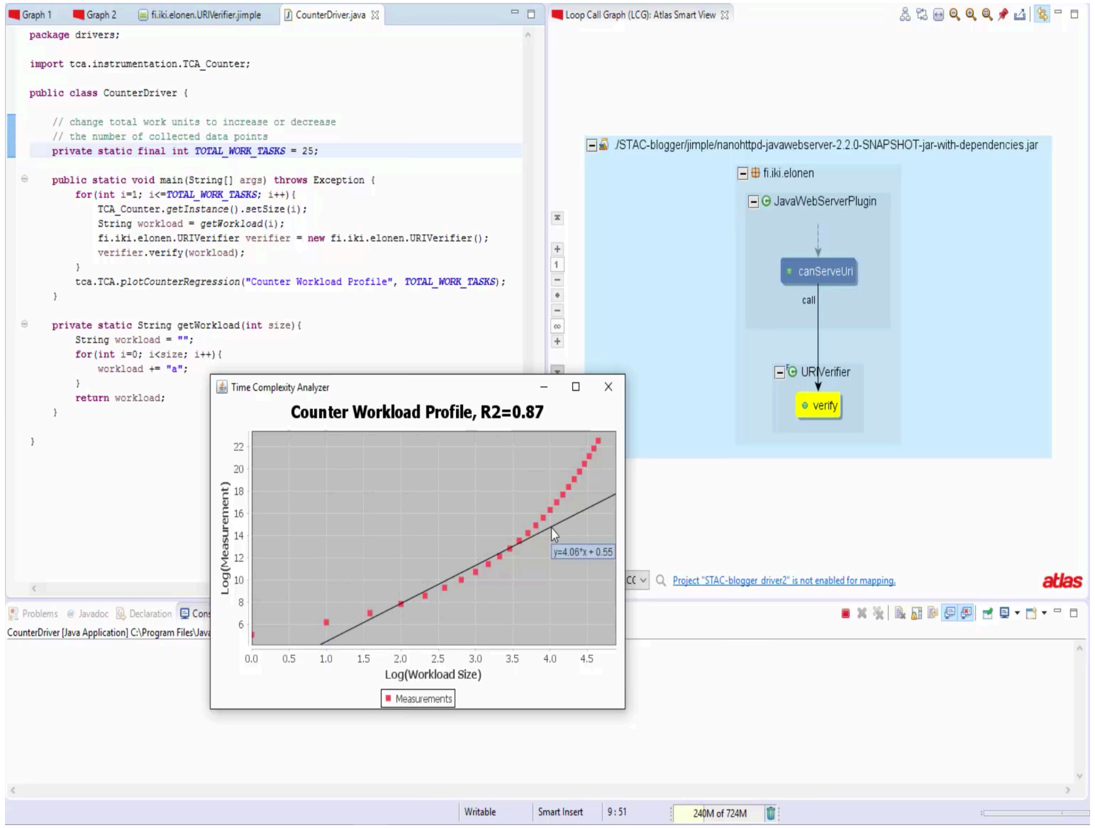
Interactive Graph Models – Loop Call Graph



Vulnerability

Time Complexity Analyzer

- Analyst picks entry point in the app using Loop Call Graph (LCG) view
  - LCG: Induced subgraph of reachable methods that contain loops
- Analyst selects methods from the LCG view to instrument
  - Probe choices: Iteration counters & Wall clock timers
- Automatic probe insertion into Jimple & reassembly into bytecode
- Automatic driver skeleton generation
  - Analyst fills in the driver with code that provides test input
- Automatic plot of the collected measurements for the given test input



TCA Instrumentation

- Iteration Counters
  - Tracks the number of times a loop header is executed
  - Platform independent, repeatable
- Wall Clock Timers
  - Uses timestamps to measure the cumulative time spent in a loop
  - More prone to noisy and inaccuracy, but still useful
    - Consider: caching or garbage collection side effects on the runtime
- Probes are inserted after selected loop headers

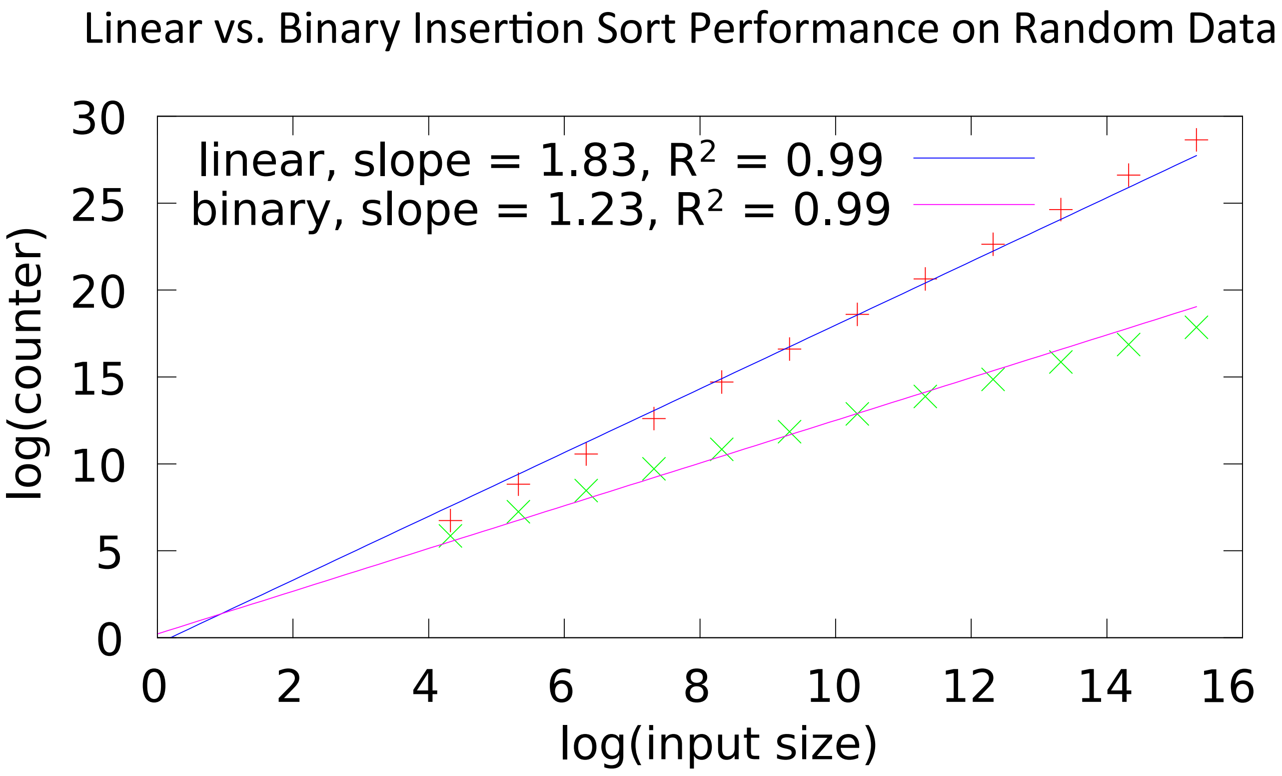
Driver Generation

- Generates driver "skeleton" with callsites to target methods
- Workload is provided by the user
  - Workload should map inputs to a "workload size"

```java
public class CounterDriver {
 private static final int TOTAL_WORK_TASKS = 30;
 public static void main(String[] args) throws
     Exception {
  for(int i=1; i<=TOTAL_WORK_TASKS; i++){
   RULER_Counter.setSize(i);
   URIVerifier verifier = new URIVerifier();
   verifier.verify(getWorkload(i));
  }
  tca.TCA.plotRegression(
   "URIVerifier.verify Workload Profile",
   TOTAL_WORK_TASKS);
 }
 private static String getWorkload(int size){
  String unit = "a";
  StringBuilder result = new StringBuilder();
  for(int i=0; i<size; i++){
   result.append(unit);
  }
  return result.toString();
 }
}
```

Complexity Analysis

- Plots results on a log-log scale
- Linear regression to fit measurements
- R^2 error value
- A slope of m on the log-log plot indicates the measured empirical complexity of n^m.
- Potential use in education for comparing empirical complexities of two algorithms



Walkthrough of Blogger

Blogger Walkthrough/Workflow

Analyst Goal

- Find most expensive loops reachable in the app
- Verify if they violate resource consumption limit within the budget

Demo: SID tools used to find AC vulnerability

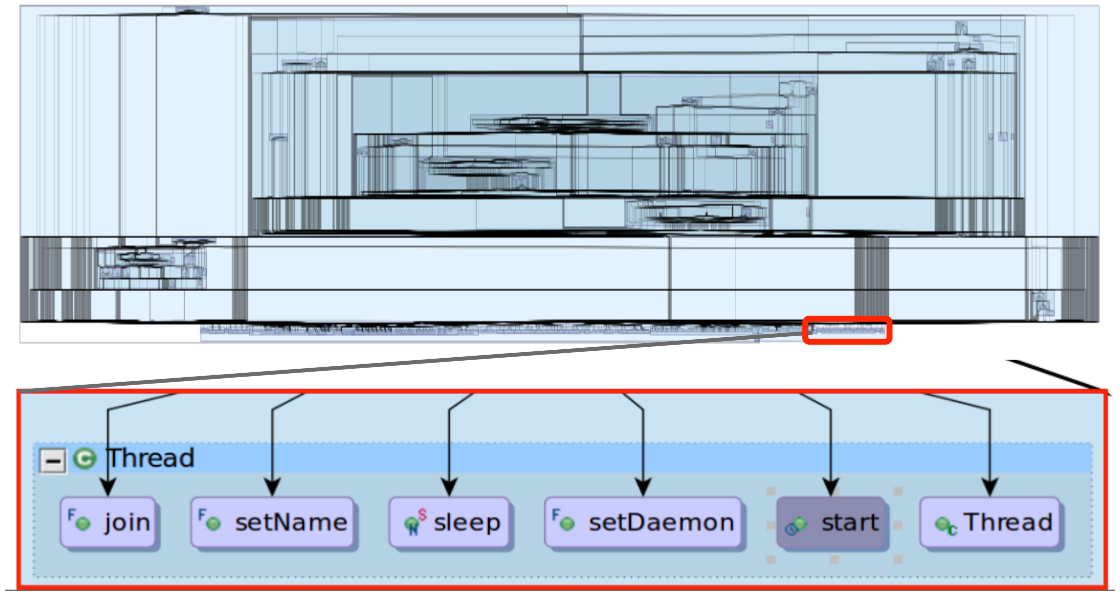
- Loop Call Graph: Find loops reachable from points of interest
- Smart Views: On-demand composable analysis
- Time Complexity Analyzer: Measure runtime performance of loops for inputs within budget

Blogger > How we found the AC vulnerability

1. Follow call graphs from entry point to code that serves client requests
   - Call graph from JavaWebServer.main() is too large
   - Notice that JDK APIs are used to start Threads
   - Look at reverse call graph from Thread.start() to see what threads are started
2. Identify use of threads in application server design
   - ServerRunnable is listener thread
   - ClientHandler is request processor thread
3. Identify loops reachable from ClientHandler using LCG
   - Narrow down scope of vulnerability to 25 of the 422 methods
4. Formulate & Validate Hypothesis
   - Run dynamic analysis informed by LCG to find method causing vulnerability

Step 1 – Locate use of Threads

Zooming into leaves of call graph from `JavaWebServer.main()` shows JDK APIs are used to start Threads



NanoHTTPD is a threaded web server.

Q. Where are threads started in the app? Which threads handle client requests?

Step 2 – ClientHandler Thread Handlers HTTP requests

ClientHandler handles client requests

Forward call graph from ClientHandler.run() is still large: 483 nodes, 1135 edges

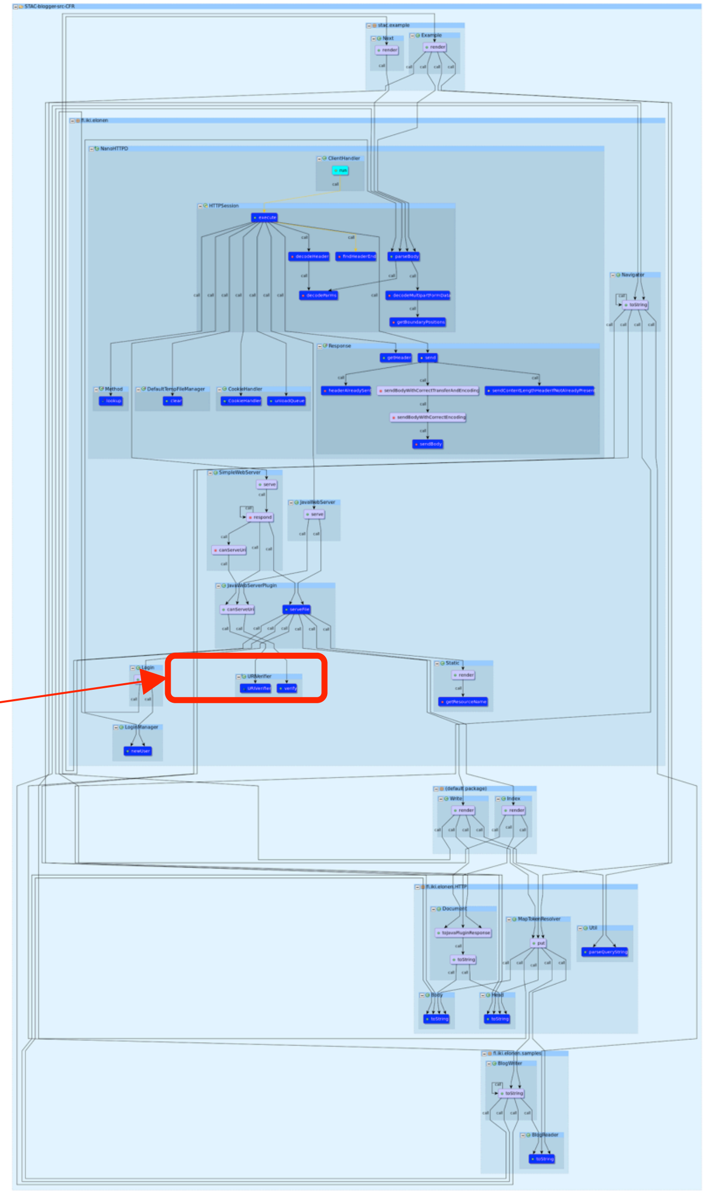
```
public class ClientHandler implements Runnable {
 ...
 @Override
 public void run() {
  // Server thread that handles client request
  OutputStream outputStream = null;
   try { ...
    HTTPSession session = new HTTPSession(...);
    while (!this.acceptSocket.isClosed())
      session.execute();
   } catch (Exception e) {...}
   finally {...}
 }
}
```

Q. What loops in the app are reachable from ClientHandler.run()?

Step 3 – Loop Call Graph

Significantly more compact view than the original call graph

- 79 nodes, 150 edges in LCG from ClientHandler.run
- 41 loops reached from ClientHandler.run
- Compared to 483 nodes, 1135 edges in the call graph
- Focuses analyst attention on loops, while preserving call reachability
- Includes the vulnerability - URIVerifier.verify()



Analyst wants to find “interesting” methods to inspect

Step 4 – Dynamic Analysis Informed by LCG

1. Analyst uses TCA to probe each of the 41 loops using Iteration Counter instrument
2. TCA compiles, runs instrumented jar (Instrumented Blogger server is started)
3. Once server is started, analyst interacts with the application using a web browser
4. TCA records the number of iterations for each loop execution

Step 4 – Dynamic Analysis Informed by LCG

Analyst issues 3 sample URLs to server

"/"

"/test"

"/stac/example/Example"

Instrumented server counts and saves # iterations for each loop exercised

2 methods record large iteration counts
- HTTPSession.findHeaderEnd()
- URIVerifier.verify()

Method Name	Iterations
NanoHTTPD.HTTPSession.findHeaderEnd.label1	4341
URIVerifier.verify.label5	2148
URIVerifier.verify.label3	1188
URIVerifier.verify.label1	1074
URIVerifier.URIVerifier.label5	270
URIVerifier.URIVerifier.label1	190
NanoHTTPD.HTTPSession.decodeHeader.Trap Region.label10	100
NanoHTTPD.Response.headerAlreadySent.label1	24
NanoHTTPD.CookieHandler.CookieHandler.label1	22
NanoHTTPD.Response.sendBody.label3	22
NanoHTTPD.HTTPSession.decodeParms.label2	16
NanoHTTPD.ClientHandler.run.Trap Region.label2	15
NanoHTTPD.Response.send.Trap Region.label6	12
NanoHTTPD.CookieHandler.unloadQueue.label1	12
NanoHTTPD.Response.getHeader.label1	12
NanoHTTPD.HTTPSession.execute.Trap Region.label6	11
NanoHTTPD.DefaultTempFileManager.clear.label1	11
NanoHTTPD.Method.lookup.label1	11
NanoHTTPD.ServerRunnable.run.Trap Region.label6	5
NanoHTTPD.start.label3	2

Step 4 – Dynamic Analysis Informed by LCG

```
private int findHeaderEnd(byte[] buf, int rlen) {
    int splitbyte = 0;
    while (splitbyte + 3 < rlen) {
        if ((buf[splitbyte] == 13) && (buf[(splitbyte + 1)] == 10) && (buf[(splitbyte + 2)] == 13) && (buf[(splitbyte + 3)] == 10)) {
            return splitbyte + 4;
        }
        splitbyte++;
    }
    return 0;
}
```

- Single loop
- Single termination condition
- Loop induction variable splitbyte:
  - Modified in one location inside the loop body
  - Monotonically increases up to termination condition

Step 4 – Dynamic Analysis Informed by LCG

```
public boolean verify(String string) {
    Tuple peek;
    LinkedList<Tuple> tuples = new LinkedList<Tuple>();
    tuples.push(new Tuple<Integer, URIElement>(0, this.verifierElements));
    while (!tuples.isEmpty() && (peek = (Tuple)tuples.pop()) != null) {
        if (((URIElement)peek.second).isFinal && ((Integer)peek.first).intValue() == string.length()) {
            return true;
        }
        if (string.length() > (Integer)peek.first) {
            for (URIElement URIElement2 : ((URIElement)peek.second).get(string.charAt((Integer)peek.first))) {
                tuples.push(new Tuple<Integer, URIElement>((Integer)peek.first + 1, URIElement2));
            }
        }
        for (URIElement child : ((URIElement)peek.second).get(-1)) {
            tuples.push(new Tuple(peek.first, child));
        }
    }
    return false;
}
```

- 3 loops
- Logic behind push and pop on loop induction variable tuples is unclear
- Analyst decides to instrument URIVerifier.verify() separately

Step 4 – Dynamic Analysis Informed by LCG

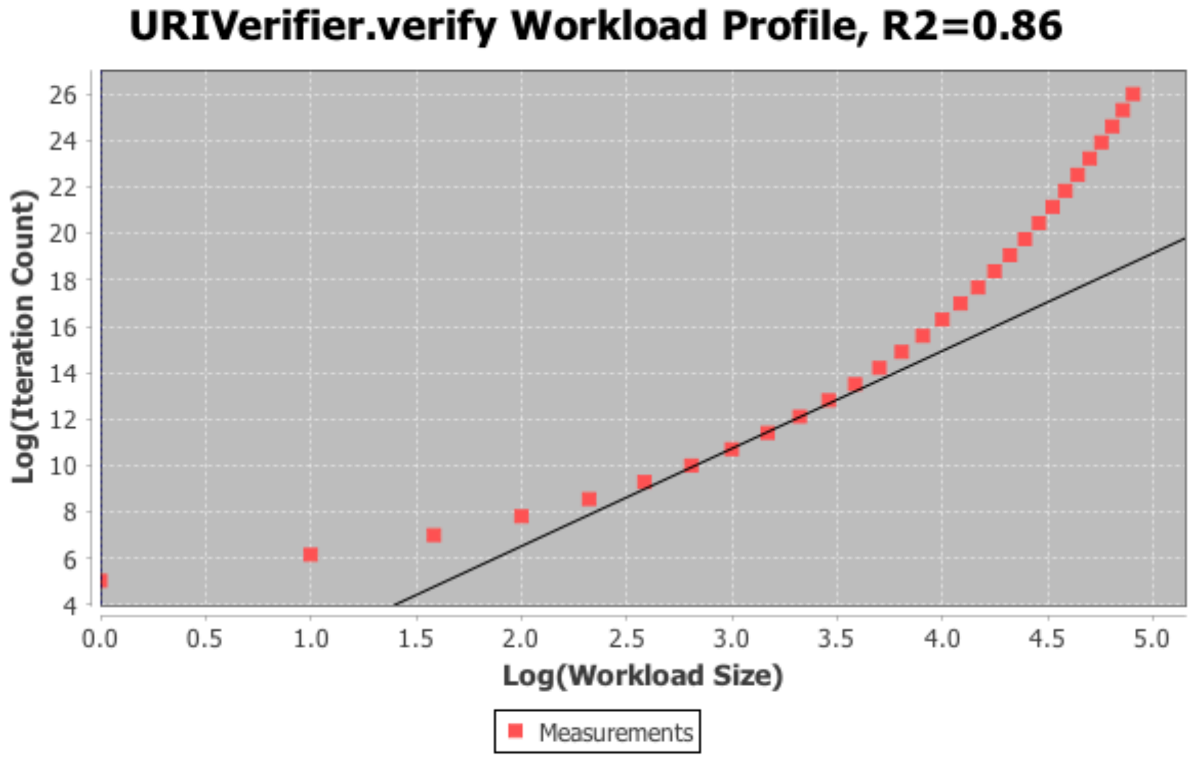
Analyst uses TCA to instrument `URIVerifier.verify()` with iteration counter Driver to test the method with URL strings of increasing length:

```
public class CounterDriver {

    private static final int TOTAL_WORK_TASKS = 30;

    public static void main(String[] args) throws Exception {
        for(int i=1; i<=TOTAL_WORK_TASKS; i++){
            RULER_Counter.setSize(i);
            URIVerifier verifier = new URIVerifier();
            verifier.verify(getWorkload(i));
        }
        tca.TCA.plotRegression("URIVerifier.verify Workload Profile", TOTAL_WORK_TASKS);
    }

    private static String getWorkload(int size){
        String unit = "a";
        StringBuilder result = new StringBuilder();
        for(int i=0; i<size; i++){
            result.append(unit);
        }
        return result.toString();
    }

}
```

Step 4 – Dynamic Analysis Informed by LCG

TCA produces a plot of # iterations in URIVerifier.verify() vs. URL string length

Analyst confirms URIVerifier.verify() exceeds budgeted time of 300 seconds for URL strings of length > 35



IOWA STATE UNIVERSITY
Department of Electrical and Computer Engineering

Tools

- SID Tools: https://ensoftcorp.github.io/SID/
  - Eclipse Plugin
  - Open Source, MIT License
  - Video Demo
- Atlas
  - Supports C/Java/JVM Bytecode (Jimple IR)
  - Free for academic use/open source projects
  - http://www.ensoftcorp.com/atlas/
- Soot
  - Bytecode to Jimple transformation
  - https://sable.github.io/soot/

Future Work

- Better heuristics to guide analyst to problem areas
  - Loops with complex termination conditions
  - Non-monotonic loops
- Thinking hard about input generation

Thank you.

- Questions?