ADELARD

Safe resource use tracking with a combination of control flow and data flow analysis

Frama-C Day 2015

Peter Bishop (pgb@adelard.com)
Sam George (srjg@adelard.com)

20 June 2016

PT/407/43140/2

Exmouth House 3–11 Pine Street London EC1R 0JH
T +44 20 7832 5850 F +44 20 7832 5853 E office@adelard.com W www.adelard.com

Overview

- Adelard background
  - Esp. assessment of smart device software for nuclear I&C
  - Our approach to assessments
- Frama-C plugins to support our work
  - Previous plug-in development
  - New plug-in development
    - Resource use tracking

Adelard – what we do

- Adelard LLP is a consultancy specialising in safety-related and safety critical computing systems
- Working in a number of industry sectors
  - Nuclear, defence, rail, aviation, medical
- Types of system
  - Smart instruments, PLCS, command and control
- Work includes
  - Safety case design and assessment
  - Software assessment
  - Assessment tools
  - Contract research
    - (UK C&I Nuclear Industry Forum - CINIF)

Adelard – what we do

- Adelard LLP is a consultancy specialising in safety-related and safety critical computing systems
- Working in a number of industry sectors
  - Nuclear, defence, rail, aviation, medical
- Types of system
  - Smart instruments, PLCS, command and control
- Work includes
  - Safety case design and assessment
  - Software assessment
  - Assessment tools
  - Contract research
    - (UK C&I Nuclear Industry Forum - CINIF)

Embedded industrial systems in UK nuclear industry

- Smart devices
  - Temperature transmitters, trip alarms, switchgear, HVAC, …
  - Configurable but not programmable – fixed firmware
  - Have a safety role

- PLCs
  - Programmable
  - Industrial control languages (like IEC 61131-3)

Role of software static analysis

- Static analysis may be used for
  - compensatory activities – show the code is acceptable despite shortcomings in the development process
  - independent confidence building

- This is a regulatory requirement for nuclear I&C (UK ONR)

Static analysis tools

- We use existing static analysis tools where available
  - LDRA TestBed
  - VectorCast
  - Frama-C (WPA)
  - Doxygen
  - ...
- But develop our own tools for specific analyses
  - Simple code parsing tools
  - Frama-C plugins

Previously developed plugins

- Partly funded by CINIF research programme
  - UK C&I Nuclear Industry Forum
- Designed to perform simple, well defined analysis
  - Coupled with manual review
- Simple Concurrency Analysis
  - Now available to all under the GNU LGPL-3 license at
    - https://bitbucket.org/adelard/simple-concurrency
- Control flow analysis
  - Will be available at the same website
  - Awaiting permission from CINIF

Concurrency Analysis

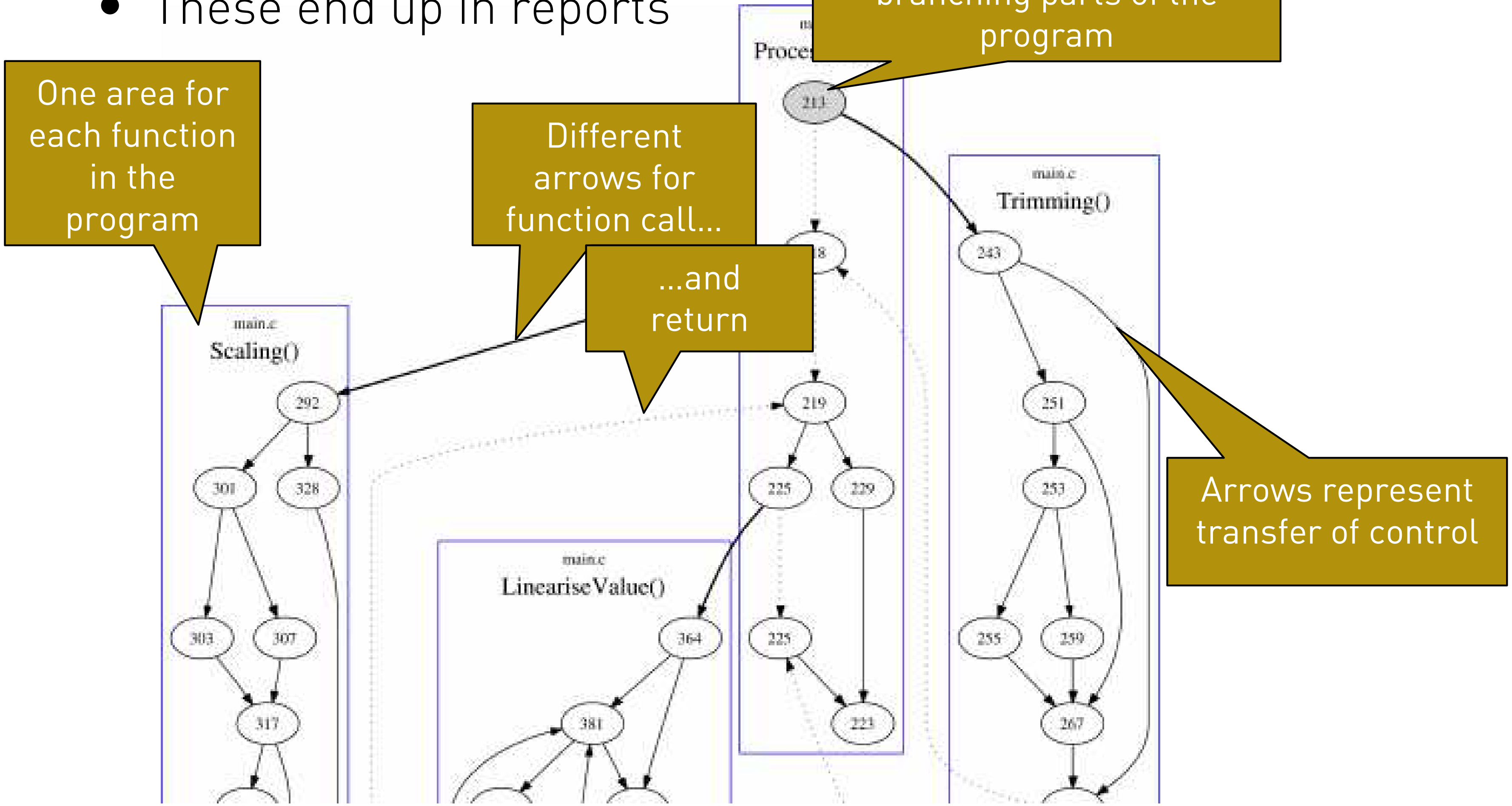
- Primarily for “main function + interrupts” software
  - But applicable to multi-tasking software designs
- Simple analysis approach
  - Identify entry points to all threads (functions with no callers)
  - Identify all static storage duration variables accessed by two or more threads
  - For each such variable, report where it is used
- Supports manual review of variable sharing
  - Indicates type and number of bits of each variable
  - Indicates whether uses are reads or writes
- Successes
  - Helped us to find real concurrency issues in several cases

Control-flow graphs

- Similar but extended version of the classic control flow graph plugin example
  - analyses for strongly connected components
  - triage for potentially dangerous kinds of unclean loops
  - standard graphviz output
  - annotation module functor
- Useful for inspecting code structure
  - Identify functions of interest
  - Broad statistics – depth and complexity of the graph
  - Look for structuring issues such as poorly-structured loops
- Allows us to quickly decide if interrupt service routines are acceptable
  - E.g., loop-free, so comparatively predictable run-time

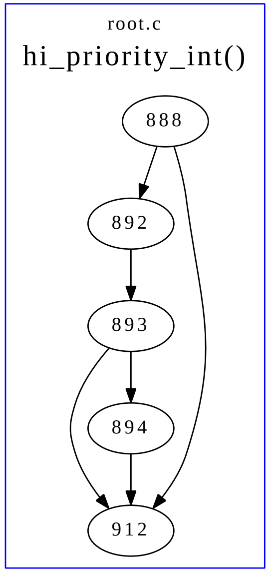
CFG output - graph

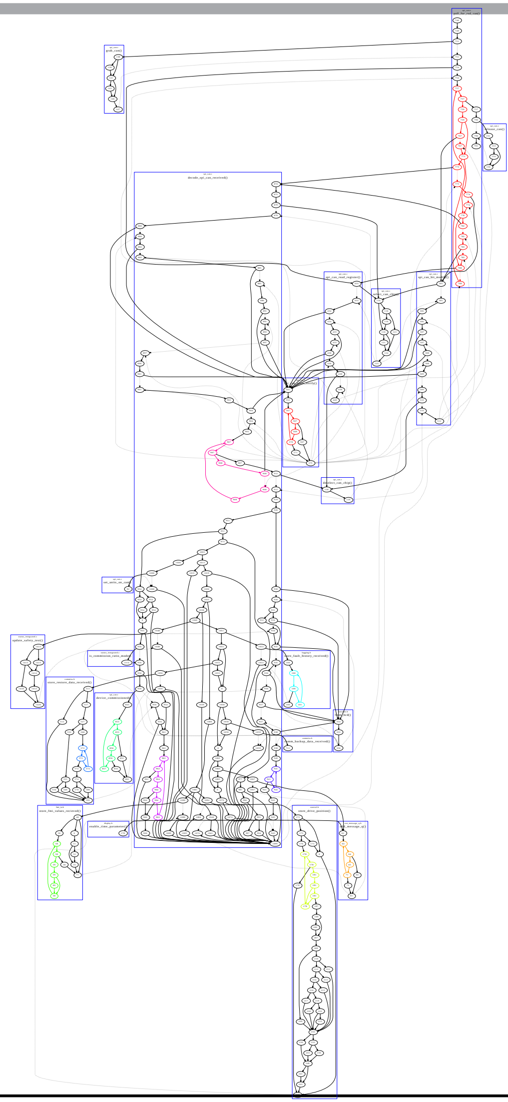
- Try to balance pretty with useful
  - These end up in reports



Finding loops

- Loops are identified as strongly-connected components in the graph (thanks, Ocamlgraph)
  - Identifies loops inside functions
  - and between functions (e.g., recursion)
- Ideally loop free in interrupt routines





Latest plugin

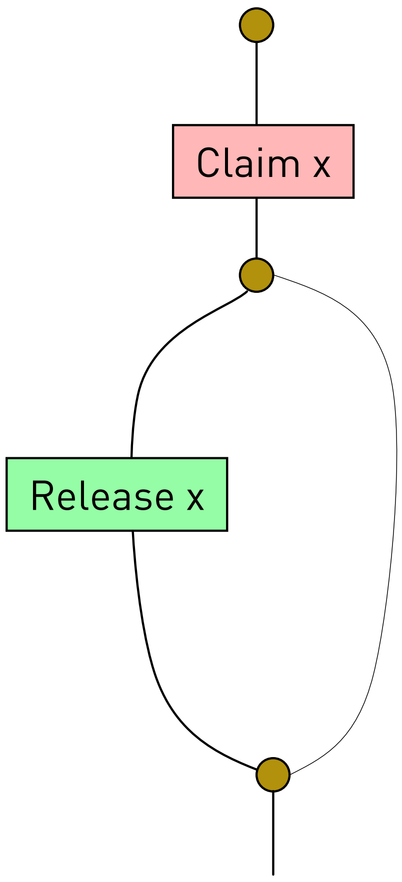
- Resource usage analysis
  - New work since 2015 Workshop
  - Still in development
- Needed for analyses of software running on a OS kernel
- Will:
  - Introduce resource usage analysis
  - Outline how the analysis is implemented in a plug-in
  - Present some examples

Resource usage analysis

- Multiple threads of execution (like OS tasks) need to claim and release resources for exclusive use, like:
  - Enable/disable interrupt
  - Claim / release semaphore
  - Claim/release heap (malloc-free)
- In a sound program thread:
  - Claim should always be followed by Release
  - No Release without a prior Claim
  - Sequence of Claims (e.g. semaphores) must be Released in reverse order (LIFO)

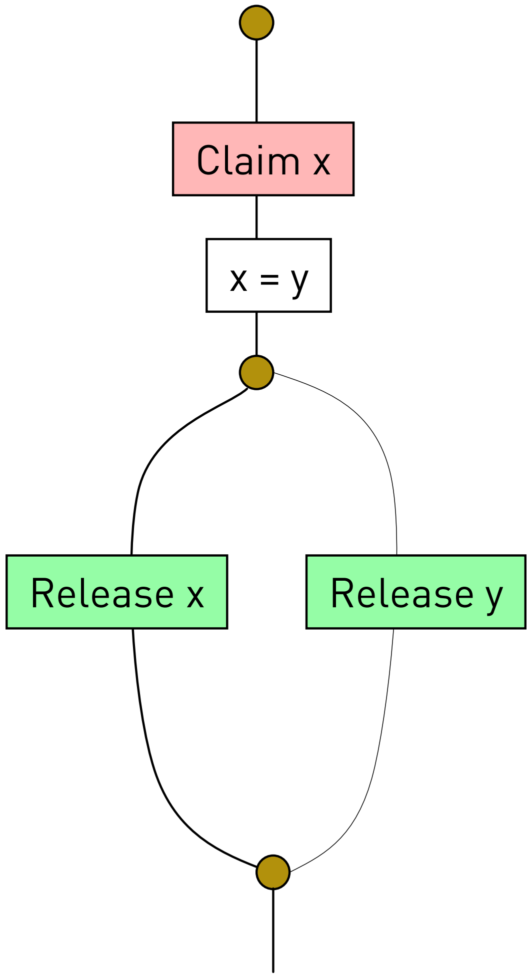
Stage 1 implementation

- Restrict resource balance analysis to single functions
  - Would be best practice software design anyway
- Scans the control flow graph for claim / release invocations on every possible control flow path (inc. loops and gotos)
- Identify any control flow sequences in function where resource controls do not balance
- Non compliance reported for manual analysis



Stage 1 Issues

- Different ways of identifying the same resource
  - constant, variable, parameter to a function, ...
- If resource identifiers do not match can incorrectly report unbalanced resource use
- Need to track all “aliases” for the same resource identifier



Plug-in implementation approach

- Most of what we are doing builds on our existing control flow plug-in
- Try to stack small independent plugins, using Frama-C memoisation
- Development issues
  - Dependency fun – using Nixos
    - Hacking the makefile:

```
substituteInPlace Makefile --replace '$(FRAMAC_SHARE)/Makefile.dynamic'
   "$TMP_MAKEFILES/Makefile.dynamic"
```

  - Plug-in API paradigms?

Resource tracking – implementation

- Immutable Ocamlgraphs
- Vertex node map - map of control flow vertices to lattice vertices
- Build up a simple semilattice, where nodes store a lattice node ID, an alias tracker structure and a reference back into the control flow graph
- History list built up recursively with function call history (needed to deal with consequences of goto)
- Fixpoint on looping
  - Checking alias maps the same (conservative)
  - Will report a problem even if a path is infeasible owing to mallocs/frees in conditionals (although well-written code should probably avoid this in practice – might be extended using complete lattice, value analysis and WP)

Malloc/free development version

- The code walks the CFG and tracks pointer aliases through a lattice. It is designed to
  - cope with reallocation of frees
  - identify resource use fixpoints on loops (conservative)
  - spot double frees
  - spot double mallocs
  - spot other masking assignments on aliasing
  - deal with convoluted control flow with gotos
  - spot resource leakage on function exit

Example code

```
#include <stdlib.h>

int f(int x);

void main() {
  int x;

  int *m = (int*) malloc(8 * sizeof(int));
  if (x) {
    x = x + 1;
    while (x<1) {
      x=f(2);
      x *= 4;
      x++;
    }
    free ((void*) m);
  }
  else {
    x=2;
    free ((void*) m);
  }
  return;
}

int f(int x) {
  return f(x*2);
}
```

Example code

```
#include <stdlib.h>

int f(int x);

void main() {
  int x;

  int *m = (int*) malloc(8 * sizeof(int));
  if (x) {
    x = x + 1;
    while (x<1) {
      x=f(2);
      x *= 4;
      x++;
    }
    free ((void*) m);
  }
  else {
    x=2;
    free ((void*) m);
  }
  return;
}

int f(int x) {
  return f(x*2);
}
```

Claim resource

Example code

```
#include <stdlib.h>

int f(int x);

void main() {
  int x;

  int *m = (int*) malloc(8 * sizeof(int));
  if (x) {
    x = x + 1;
    while (x<1) {
      x=f(2);
      x *= 4;
      x++;
    }
    free ((void*) m);
  }
  else {
    x=2;
    free ((void*) m);
  }
  return;
}

int f(int x) {
  return f(x*2);
}
```

Claim resource

Release resource

Release resource

Analysis Output

- Normally,

```
[cfganalysis] Computing CFG
[cfganalysis] Starting malloc/free resource tracking analysis
[cfganalysis] Starting analysis of function "f"
[cfganalysis] Analysis of function "f" completed without errors
[cfganalysis] Starting analysis of function "main"
[cfganalysis] Analysis of function "main" completed without errors
[cfganalysis] malloc/free resource track analysis finished for all functions.
```

- Finds “malloc”
- Finds “free” on both paths (line 16 and 20)
- Balance achieved

Analysis Output

- If we remove one of the `frees`:

```
[cfganalysis] Computing CFG
[cfganalysis] Starting malloc/free resource tracking analysis
[cfganalysis] Starting analysis of function "f"
[cfganalysis] Analysis of function "f" completed without errors
[cfganalysis] Starting analysis of function "main"
[cfganalysis] Warning: analysis of function "main" completed but with resource leakage
[cfganalysis] malloc/free resource track analysis finished for all functions.
```

- Finds “malloc”
- But balance not achieved on both paths

Analysis Output

- If we add another `free`:

```
[cfganalysis] Computing CFG
[cfganalysis] Starting malloc/free resource tracking analysis
[cfganalysis] Starting analysis of function "f"
[cfganalysis] Analysis of function "f" completed without errors
[cfganalysis] Starting analysis of function "main"
[cfganalysis] In analysis of vertex test3a.c:22:
[cfganalysis] - Dangerous resource release from m
[cfganalysis] Trace details:
[cfganalysis] - test3a.c:20 (return): No resources allocated
[cfganalysis] - test3a.c:20: No resources allocated
[cfganalysis] - test3a.c:19: Resource ID 0 [m]
[cfganalysis] - test3a.c:9: Resource ID 0 [m]
[cfganalysis] - test3a.c:8 (return): Resource ID 0 [m]
[cfganalysis] - test3a.c:8: Resource ID 0 [m]
[cfganalysis] Analysis of function "main" terminated with errors
[cfganalysis] malloc/free resource track analysis finished for all functions.
```

- Finds “malloc”
- But balance not achieved on both paths

Fixpoint example

```
#include <stdlib.h>

int f(int x);

void main() {
  int x;

  int *m = (int*) malloc(8 * sizeof(int));
  if (x) {
    x = x + 1;
    while (x<1) {
      free ((void*) m);
      x = f(2);
      x *= 4;
      x++;
      m = (int*) malloc(8 * sizeof(int));
    }
    free ((void*) m);
  }
  else {
    x=2;
    free ((void*) m);
  }
  /* free ((void*) m); */
  return;
}

int f(int x) {
  return f(x*2);
}
```

```
[cfganalysis] Computing CFG
[cfganalysis] Starting malloc/free resource tracking analysis
[cfganalysis] Starting analysis of function "f"
[cfganalysis] Analysis of function "f" completed without errors
[cfganalysis] Starting analysis of function "main"
[cfganalysis] Analysis of function "main" completed without errors
[cfganalysis] malloc/free resource track analysis finished for all functions.
```

Fixpoint example 2

```
#include <stdlib.h>

int f(int x);

void main() {
  int x;

  int *m = (int*) malloc(8 * sizeof(int));
  int *n = m;
  if (x) {
    x = x + 1;
    while (x<1) {
      free((void*) m);
      x=f(2);
      x *= 4;
      x++;
      n = (int*) malloc(8 * sizeof(int));
      m = n;
      x *= 4;
    }
    free ((void*) m);
  }
  else {
    x=2;
    free ((void*) m);
  }
  return;
}

int f(int x) {
  return f(x*2);
}
```

```
[cfganalysis] Computing CFG
[cfganalysis] Starting malloc/free resource tracking analysis
[cfganalysis] Starting analysis of function "f"
[cfganalysis] Analysis of function "f" completed without errors
[cfganalysis] Starting analysis of function "main"
[cfganalysis] Analysis of function "main" completed without errors
[cfganalysis] malloc/free resource track analysis finished for all functions.
```

And if we remove assignment:

```
[cfganalysis] Computing CFG
[cfganalysis] Starting malloc/free resource tracking analysis
[cfganalysis] Starting analysis of function "f"
[cfganalysis] Analysis of function "f" completed without errors
[cfganalysis] Starting analysis of function "main"
[cfganalysis] In analysis of vertex test3d.c:12:
[cfganalysis] - Failed to find resource use fixpoint on looping
[cfganalysis] Trace details:
[cfganalysis] - test3d.c:17: Resource ID 1 [n]
[cfganalysis] - test3d.c:16: No resources allocated
[cfganalysis] - test3d.c:15: No resources allocated
[cfganalysis] - test3d.c:14 (return): No resources allocated
[cfganalysis] - test3d.c:14: No resources allocated
[cfganalysis] - test3d.c:13 (return): No resources allocated
[cfganalysis] - test3d.c:13: No resources allocated
[cfganalysis] - test3d.c:13: Resource ID 0 [m, n]
[cfganalysis] - test3d.c:12: Resource ID 0 [m, n]
[cfganalysis] - test3d.c:12: Resource ID 0 [m, n]
[cfganalysis] - test3d.c:11: Resource ID 0 [m, n]
[cfganalysis] - test3d.c:10: Resource ID 0 [m, n]
[cfganalysis] - test3d.c:9: Resource ID 0 [m, n]
[cfganalysis] - test3d.c:8 (return): Resource ID 0 [m]
[cfganalysis] - test3d.c:8: Resource ID 0 [m]
[cfganalysis] Analysis of function "main" terminated with errors
[cfganalysis] malloc/free resource track analysis finished for all functions.
```

Or add an extra `free`:

```
[cfganalysis] Computing CFG
[cfganalysis] Starting malloc/free resource tracking analysis
[cfganalysis] Starting analysis of function "f"
[cfganalysis] Analysis of function "f" completed without errors
[cfganalysis] Starting analysis of function "main"
[cfganalysis] In analysis of vertex test3e.c:25:
[cfganalysis] - Dangerous resource release from m
[cfganalysis] Trace details:
[cfganalysis] - test3e.c:20 (return): No resources allocated
[cfganalysis] - test3e.c:20: No resources allocated
[cfganalysis] - test3e.c:12: Resource ID 0 [m, n]
[cfganalysis] - test3e.c:12: Resource ID 0 [m, n]
[cfganalysis] - test3e.c:12: Resource ID 0 [m, n]
[cfganalysis] - test3e.c:11: Resource ID 0 [m, n]
[cfganalysis] - test3e.c:10: Resource ID 0 [m, n]
[cfganalysis] - test3e.c:9: Resource ID 0 [m, n]
[cfganalysis] - test3e.c:8 (return): Resource ID 0 [m]
[cfganalysis] - test3e.c:8: Resource ID 0 [m]
[cfganalysis] Analysis of function "main" terminated with errors
[cfganalysis] malloc/free resource track analysis finished for all functions.
```

An example with a `goto`:

```c
#include <stdlib.h>

int f(int x);

void main() {
  int x;
  int *m = (int*) malloc(8 * sizeof(int));
  int *n = m;
  if (x) {
    x = x + 1;
    while (x<1) {
      free((void*) m);
      x = f(2);
      x *= 4;
      x++;
m2:   n = (int*) malloc(8 * sizeof(int));
      m = n;
    }
    free ((void*) n);
  }
  else {
    x = 2;
    goto m2;
  }
  free ((void*) m);
  return;
}

int f(int x) {
  return f(x*2);
}
```

```
[cfganalysis] Computing CFG
[cfganalysis] Starting malloc/free resource tracking analysis
[cfganalysis] Starting analysis of function "f"
[cfganalysis] Analysis of function "f" completed without errors
[cfganalysis] Starting analysis of function "main"
[cfganalysis] In analysis of vertex test3f.c:16:
[cfganalysis] - Pointer n already assigned in context
[cfganalysis] Trace details:
[cfganalysis] - test3f.c:22: Resource ID 0 [m, n]
[cfganalysis] - test3f.c:9: Resource ID 0 [m, n]
[cfganalysis] - test3f.c:8: Resource ID 0 [m, n]
[cfganalysis] - test3f.c:7 (return): Resource ID 0 [m]
[cfganalysis] - test3f.c:7: Resource ID 0 [m]
[cfganalysis] Analysis of function "main" terminated with errors
[cfganalysis] malloc/free resource track analysis finished for all functions.
```

And finally:

```c
#include <stdlib.h>

int f(int x);

void main() {
  int x;
  int *m = (int*) malloc(8 * sizeof(int));
  int *n = m;
  if (x) {
    x = x + 1;
    while (x<1) {
      free((void*) m);
      x = f(2);
      x *= 4;
      x++;
m2:   n = (int*) malloc(8 * sizeof(int));
      m = n;
    }
    free ((void*) n);
  }
  else {
    x = 2;
    free ((void*) m);
    m = (int*) malloc(8 * sizeof(int));
    goto m2;
  }
  free ((void*) m);
  return;
}

int f(int x) {
  return f(x*2);
}
```

```
[cfganalysis] Computing CFG
[cfganalysis] Starting malloc/free resource tracking analysis
[cfganalysis] Starting analysis of function "f"
[cfganalysis] Analysis of function "f" completed without errors
[cfganalysis] Starting analysis of function "main"
[cfganalysis] In analysis of vertex test3g.c:17:
[cfganalysis] - Live pointer m masked
[cfganalysis] Trace details:
[cfganalysis] - test3g.c:16 (return): Resource ID 1 [m] Resource ID 2 [n]
[cfganalysis] - test3g.c:16: Resource ID 1 [m] Resource ID 2 [n]
[cfganalysis] - test3g.c:25: Resource ID 1 [m]
[cfganalysis] - test3g.c:24 (return): Resource ID 1 [m]
[cfganalysis] - test3g.c:24: Resource ID 1 [m]
[cfganalysis] - test3g.c:23 (return): No resources allocated
[cfganalysis] - test3g.c:23: No resources allocated
[cfganalysis] - test3g.c:22: Resource ID 0 [m, n]
[cfganalysis] - test3g.c:9: Resource ID 0 [m, n]
[cfganalysis] - test3g.c:8: Resource ID 0 [m, n]
[cfganalysis] - test3g.c:7 (return): Resource ID 0 [m]
[cfganalysis] - test3g.c:7: Resource ID 0 [m]
[cfganalysis] Analysis of function "main" terminated with errors
[cfganalysis] malloc/free resource track analysis finished for all functions
```

Summary

- We perform software assessment for the nuclear industry
  - simple smart devices
  - more complex OS kernel-based products
- Frama C is a useful tool for supporting the these assessments
  - Standard plug-ins
  - Special plug-ins where necessary
- Policy of open-sourcing our mature plug-ins
- Resource usage claim release plug-in
  - Initially for Heap use
- Development ongoing
  - Extension to different resource types
  - Resource balance over whole program if unbalanced function
  - User-friendly interface using GUI

- Thank you for your attention

- Any questions?

ADELARD