Contract-Based Verification of Timing Enforcers

Sagar Chaki, Dionisio de Niz

October 6, 2016

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.
REV-03.18.2016.0

Copyright 2016 Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

DM-0004050

Motivation

STAC = software that accesses the system clock, exchanges clock values, and uses these values to set timers and perform computation

- Key to real-time and cyber-physical systems
- Essential to keep software in sync with the physical world
- Examples = thread schedulers and time budget enforcers, distributed protocols (e.g., plug-and-play medical devices)

Goal : Formally verify STACs at the source code level using deductive (aka auto-active) verification

- Target: ZSRM mixed-criticality scheduler
  - Performs thread CPU allocation and time budget enforcement
  - Available as Linux kernel module implemented in C
  - Currently we focus on ZSRM budget enforcement only

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Why Verify Source Code?

Push assurance closer to executable level

- Use verified compilers (e.g., CompCERT) to close the final gap

Don’t need to sacrifice performance

- This is a problem when we verify models
- And is a no-go for low-level system software

Easier to integrate with existing systems

- Linux kernel module means anyone using Linux can use it
- Can be modified to work with other OSes, such as SEL4
- What You Verify Is What You Execute!

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Why use Auto-Active Verification?

Picture (optional)

Soundness

Language expressivity

- Pointers, recursion, loops

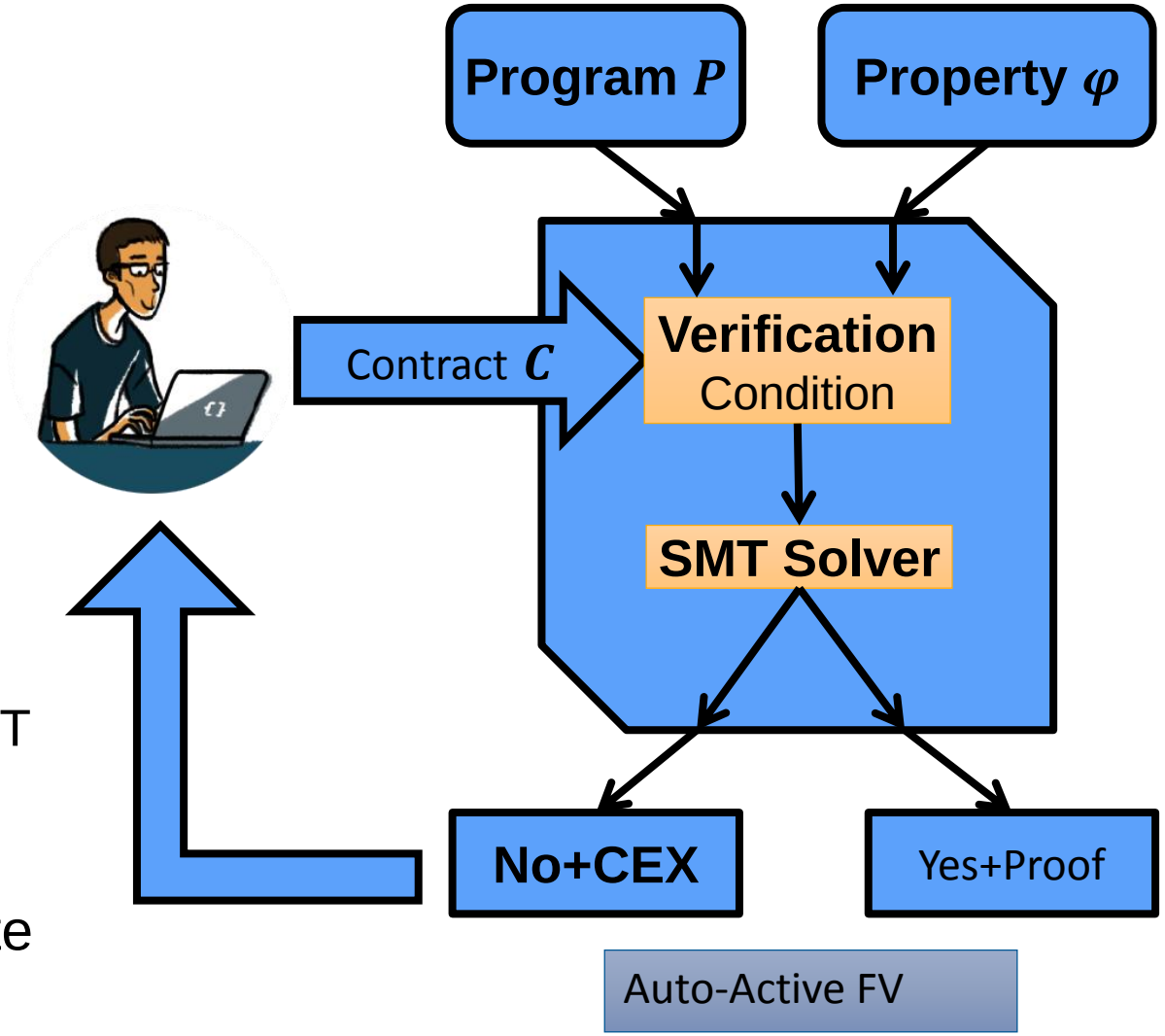
Rich specification

- Quantifiers
- Predicates
- Separation

Tool maturity

- Frama-C
  - Multiple backend SMT solvers

Good Balance between human intuition and brute force search



© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

FRAMA-C

Deductive verifier for ANSI C programs

- https://frama-c.com/
- Used to verify software in a number of projects (NASA, avionics, container libraries, hypervisors)

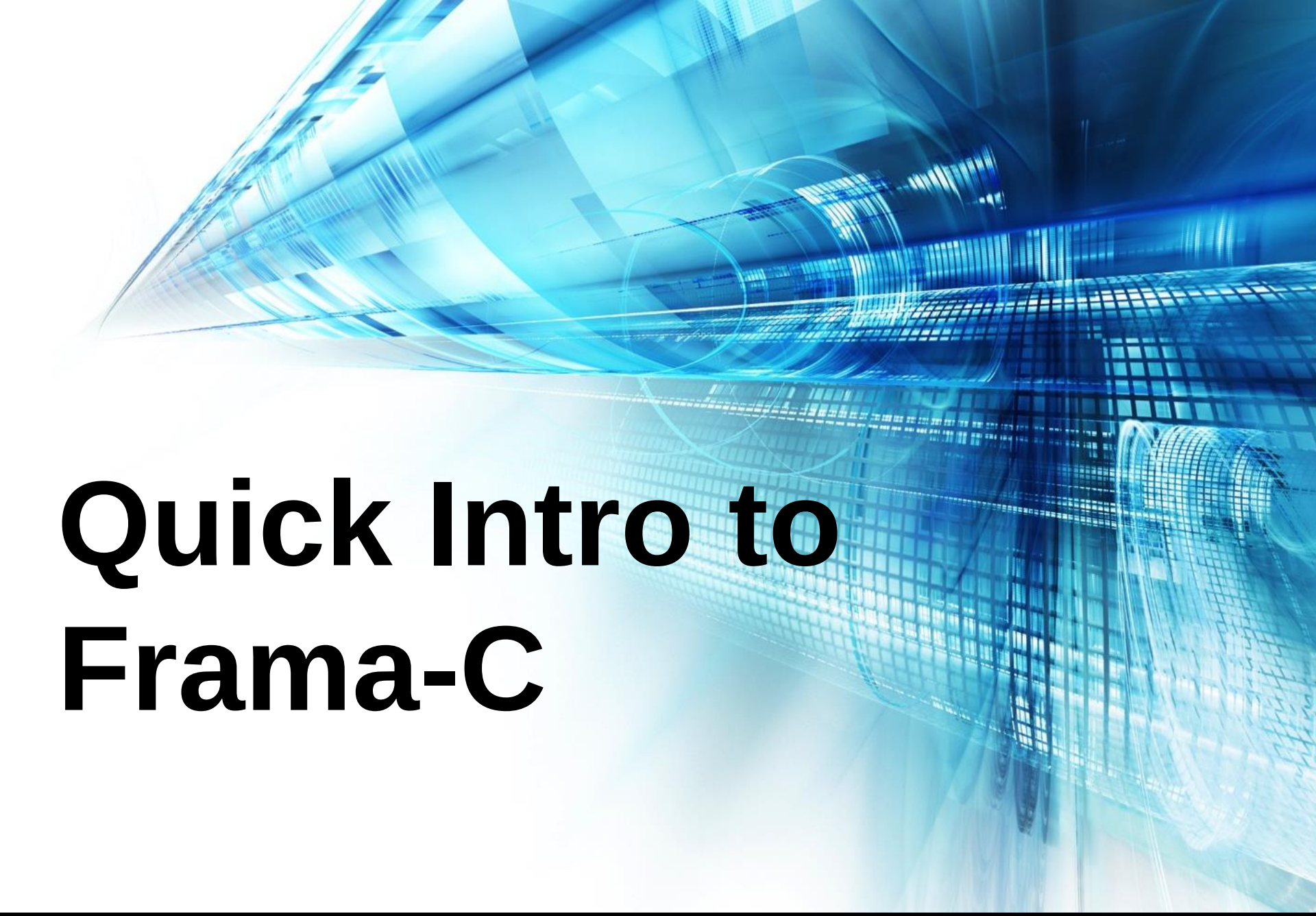
User provides function contracts and loop invariants

- Pre-and-post conditions, which variables are modified, etc.
- Expressed via ACSL language
  - https://frama-c.com/download/acsl-implementation-Aluminium-20160501.pdf
  - http://www.dcc.fc.up.pt/~nam/aulas/0910/vfs/teoricas/acsl-tutorial.pdf

Frama-C incorporates a number of plugins

- We use the WP (weakest-precondition) plugin

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.



Quick Intro to Frama-C

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

ZSRM Timing Enforcer Terminology

Threads/tasks

- $T = \{\tau_1, \tau_2, \ldots\}$
- Executes with preemption (i.e., broken up into chunks)
- Each task is a periodic sequence of “jobs”

System calls

- Job arrives : $job\_arrive(\tau)$
- Job departs : $job\_depart(\tau)$

Timer handlers

Enforcer Functions EF = System calls ∪ Timer handlers

- Execute atomically (i.e., without preemption)

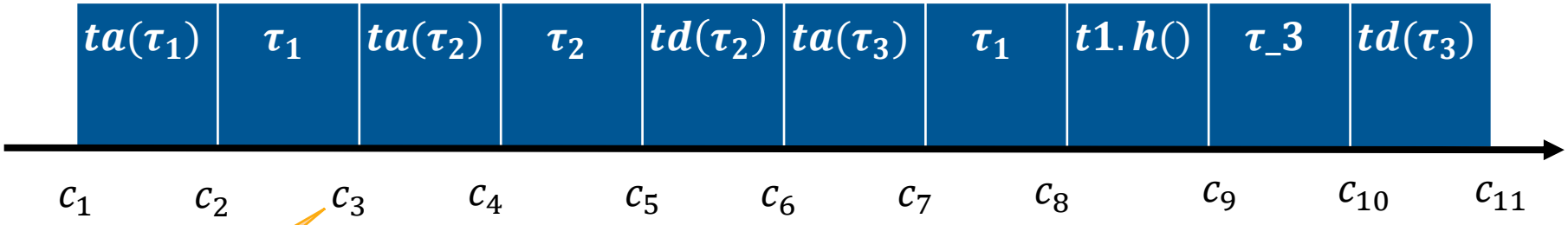
© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Execution/Timeline

Picture (optional)

Time = Global "Newtonian" clock

- Flows monotonically, dense real-time



Timestamps

$$\text{Execution } \pi = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 \ldots s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$$

$$\text{State } s_i = (c_i, variable\ values)$$

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

General Verification Strategy

Express property to be verified as an ACSL predicate:

```
/*@predicate prop = …;*/
```

For each enforcer function, define an ACSL contract that assumes and ensures “prop”

```
/*@requires prop;
  @assigns …;
  @ensures prop;*/
```

For initialization function, we only add “@ensures prop”

Verify the contracts using Frama-C

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Implementation Details

In ZSRM, each task is a reservation. Inspired by resource kernels, such as Linux-RK.

The “reserve” struct has pointer fields that enable construction and manipulation of linked lists.

Memory for reservations allocated statically. Simplifies verification considerably since “separation” between the array elements is easy to specify and sufficient for verification.

```
struct reserve {
  pid_t  pid;
  int rid;
  unsigned long long start_ns;
  unsigned long long stop_ns;
  unsigned long long current_exectime_ns;
  unsigned long long exectime_ns;
  struct timespec period;
  unsigned long long period_ns;
  struct timespec execution_time;
  int priority;
  struct zs_timer period_timer;
  struct zs_timer enforcement_timer;
  struct reserve *next;
  struct reserve *rm_next;
} reserve_table[MAX_RESERVES];
```


© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Basic Predicates

– A pointer is either the address of an element of `reserve_table` or `NULL`.

```
/*@predicate elem(struct reserve *p) = \exists int i;
(0 <= i < 10 && p == &(reserve_table[i]));*/
/*@predicate elemNull(struct reserve *p) =
(p == \null) || elem(p);*/
```

– Variables `readyq` and `rm_head` are either the address of an element of `reserve_table` or `NULL`.

```
/*@predicate fp11 = elemNull(readyq);*/
/*@predicate fp12 = elemNull(rm_head);*/
```

– The `rid` fields of `reserve_table` elements are correct (note that `==>` denotes logical implication in ACSL):

```
/*@predicate fp14 = \forall int i; 0 <= i < 10 ==>
  reserve_table[i].rid == i;*/
```


© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Linked List Predicates

Two pointer fields used to maintain separate linked lists:
- next = list of "ready" tasks in order of decreasing priority
- rm_next = list of all tasks maintained in order of increasing period

Pointers are either NULL or point to appropriate array elements.

```
/*@predicate fp21 = \forall struct reserve *p;
elem(p) ==> elemNull(p->next);*/
/*@predicate fp22 = \forall struct reserve *p;
elem(p) ==> elemNull(p->rm_next);*/
```

List of ready tasks maintained in order of decreasing priority

```
/*@predicate fp31 = \forall struct reserve *p, *q;
elem(p) ==> elem(q) ==> (p->next == q) ==>
(p->priority >= q->priority);*/
```

List of ready tasks maintained in order of increasing period

```
/*@predicate fp32 = \forall struct reserve *p, *q;
elem(p) ==> elem(q) ==> (p->rm_next == q) ==>
(p->period_ns <= q->period_ns);*/
```


© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Time-Related Predicates


```
/*@predicate zsrm3 = \forall int i; 0 <= i < 10 ==>
  reserve_table[i].current_exectime_ns <=
  reserve_table[i].exectime_ns;*/
```

Time Used

Time Budget

```
/*@predicate zsrm2 = elem(readyq) ==>
  (readyq->enforcement_timer.expiration.tv_sec
    * 1000000000L +
  readyq->enforcement_timer.expiration.tv_nsec) <=
  (readyq->exectime_ns - readyq->current_exectime_ns);*/
```

Timer Offset

Time Left in Budget

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Using Ghost Variable to track execution time

We add a ghost field $real\_exectime\_ns$ to the $reserve$ struct

- This is suppose to compute the actual CPU usage of task
- We update this variable in $job\_arraive()$ and $job\_depart()$

We prove that the time accounting by ZSRM source code is conservative, i.e., $current\_exectime\_ns \geq real\_exectime\_ns$

```
/*@predicate zsrm1 = \forall int i; 0 <= i < 10 ==>
  reserve_table[i].real_exectime_ns <=
  reserve_table[i].current_exectime_ns;*/
```

Overall predicate is the conjunction of all the predicates seen so far

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Final Verification

Frama-C Aluminium

- 3 backend SMT solvers : Z3, CVC3, CVC4
- 200s timeout

Dell Blade Server

- 40 cores (20 × 2 hyper-threads)
  - Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz
- 128GB RAM
- Ubuntu 14.04

ZSRM implemented in C as user-space program

- 1484 LOC (14549 LOC after preprocessing)
- 33 functions (enforcer functions and others called by them)

Total verification time : approx 18 minutes (real) 25 minutes (user)

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Conclusion and Future Work

Using deductive verification to prove timing and logical correctness of the ZSRM timing enforcer

- Prove that threads are restricted to CPU budgets
- Formalized property as ACSL contracts
- Used Frama-C to discharge them

Preliminary work but promising results

- Working on formalizing soundness of approach
- Completing implementation of kernel module
- Verification of additional timing property (e.g., zero-slack instant)

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

© 2016 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.