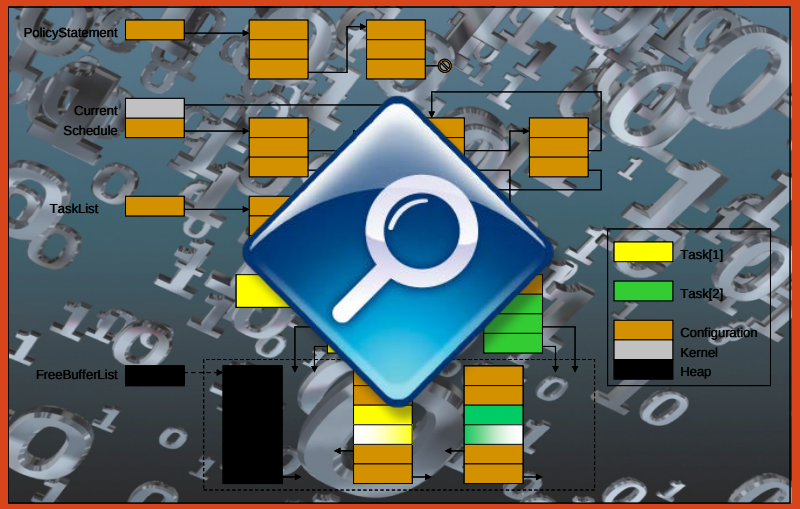
Data Flow Analysis of a Xen-based Separation Kernel

David Greve – Rockwell Collins
Steven VanderLeest - Dornerworks
09 December 2013



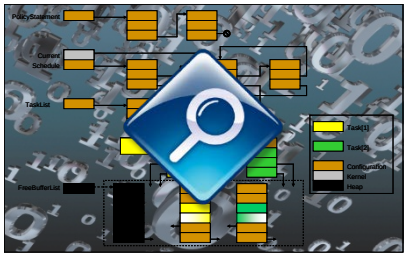
Research sponsored by Space and Naval Warfare Systems Command
Contract N66001-12-C-5221
Approved for public release; distribution unlimited.
© 2013 Rockwell Collins and Dornerworks, All rights reserved.

Program Overview

- Dornerworks
  - Developing Xen-based Separation Kernel
  - ARLX
    - **A**RINC 653 **R**eal-time **L**inux on **X**en
  - Safety and Security
    - DO-178C Level A, CC EAL6
- Rockwell Collins
  - Subcontractor under a Navy SBIR Topic "Isolation Techniques for Untrusted Software"
  - Evaluating the use of the DFL Framework
    - Means of Generating Formal Evidence of Separation
- Objectives of Evaluation
  - Model and Analyze Scheduling Subsystem (TOE)
    - a653_do_schedule()
  - Establish Viability of DFL Approach
  - Identify Residual Issues

DORNERWORKS

Rockwell Collins
Building trust every day



DornerWorks

Where Software & Hardware Design Meet

Engineering design services (electronic hardware, embedded software, FPGA logic design) for three markets:

- Aerospace & Defense
- Medical
- Automotive and ruggedized industrial

Small business located in Michigan with 80 employees (mostly electrical, computer, and software engineers).

AS9100, ISO9001, ISO13485 certified
ITAR Compliant

Technical expertise:

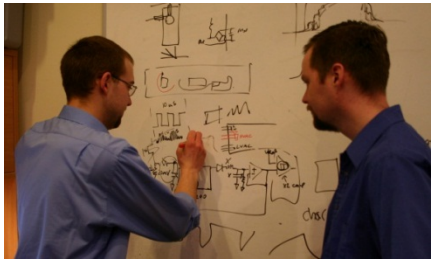
- RTOS Board Support Packages (BSP) and device drivers
- Real-time control
- FPGA logic, especially video
- DO-178 and DO-254 rigorous design practices

Open Source Business Model

- Split OSS/Commercial Product
  - Open Source Software
  - Proprietary Extensions, Tools, and Artifacts

- Platform Support
  - Engineering Design Services
  - Board Support Packages

- Product Specialists
  - Training
  - Support



Xen

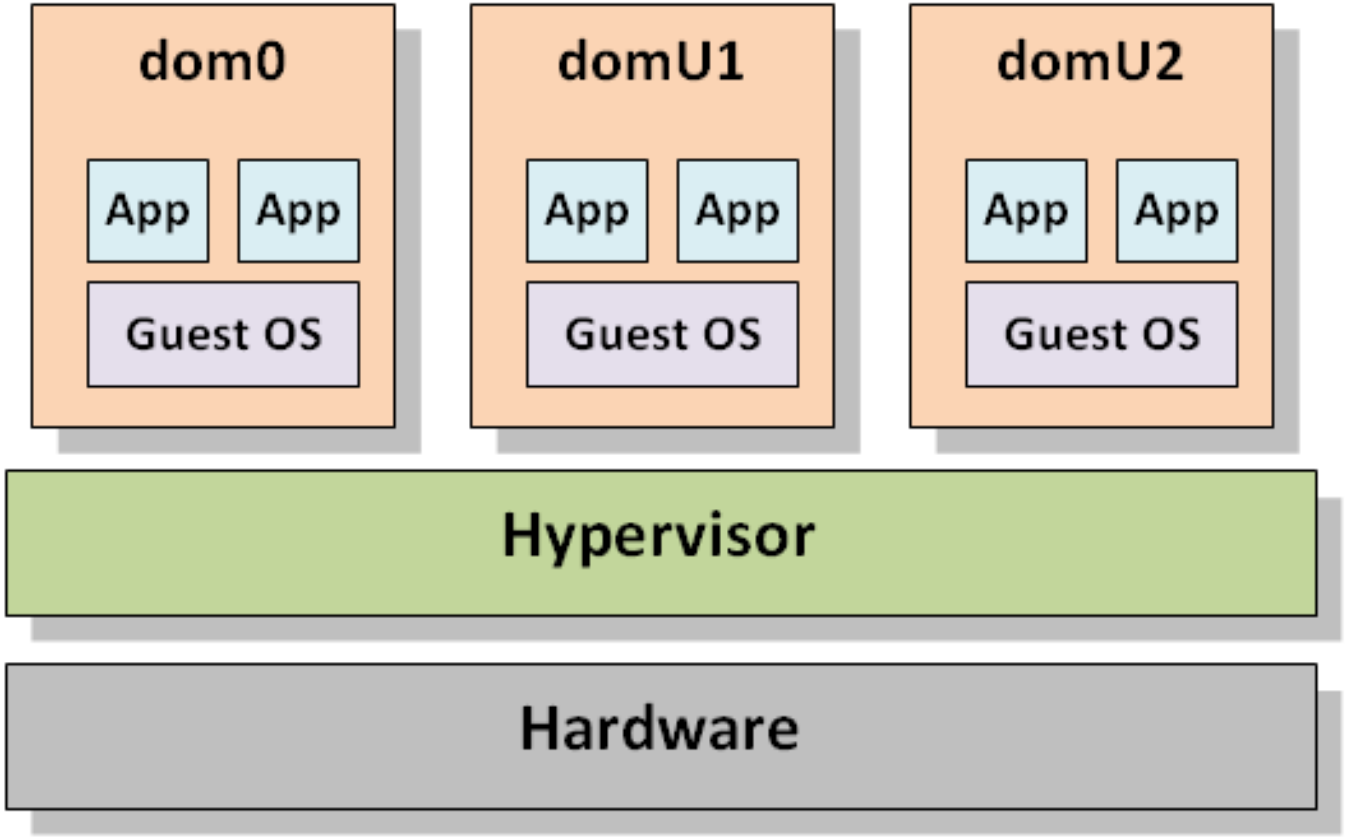
- Type 1 Hypervisor (Virtual Machine Monitor)
  - Runs directly on hardware
  - Virtualizes Hardware Platform
  - Supports Multiple Operating Systems

- Limited Capabilities
  - Scheduling
  - Memory Protection
  - I/O and Inter-VM communication

- Open Source
  - GPLv2

ARLX

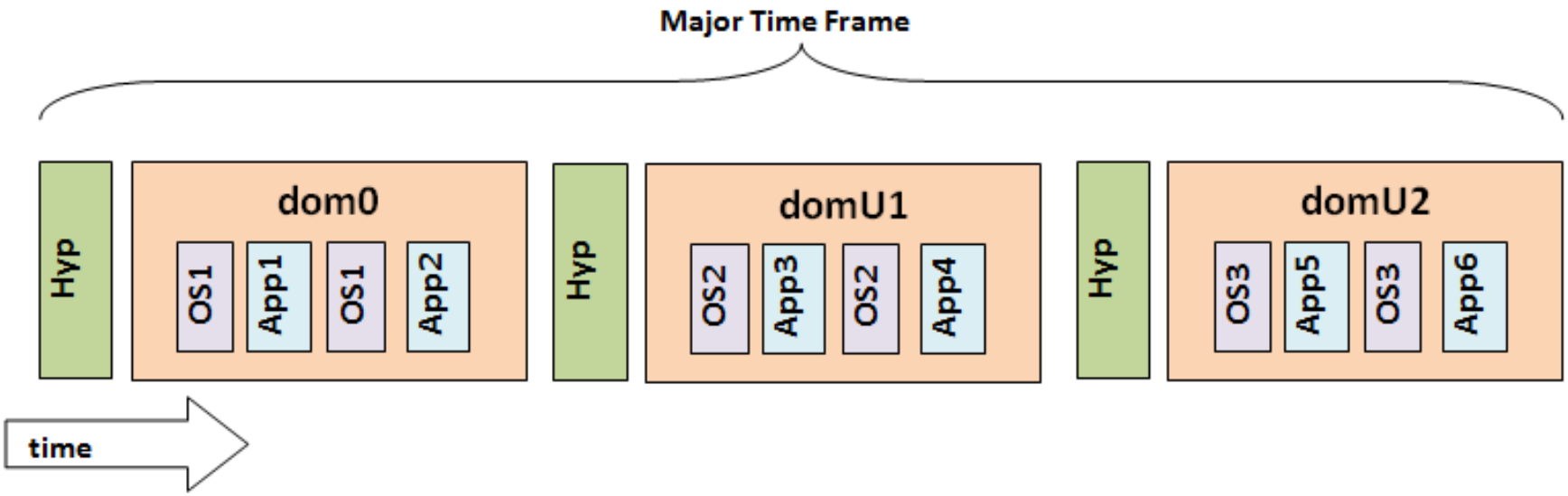
- ARLX: ARINC 653 Real-time Linux on Xen
  - Safe and Secure Separation Kernel Hypervisor
  - Xen Domains as Partitions



ARLX Scheduling

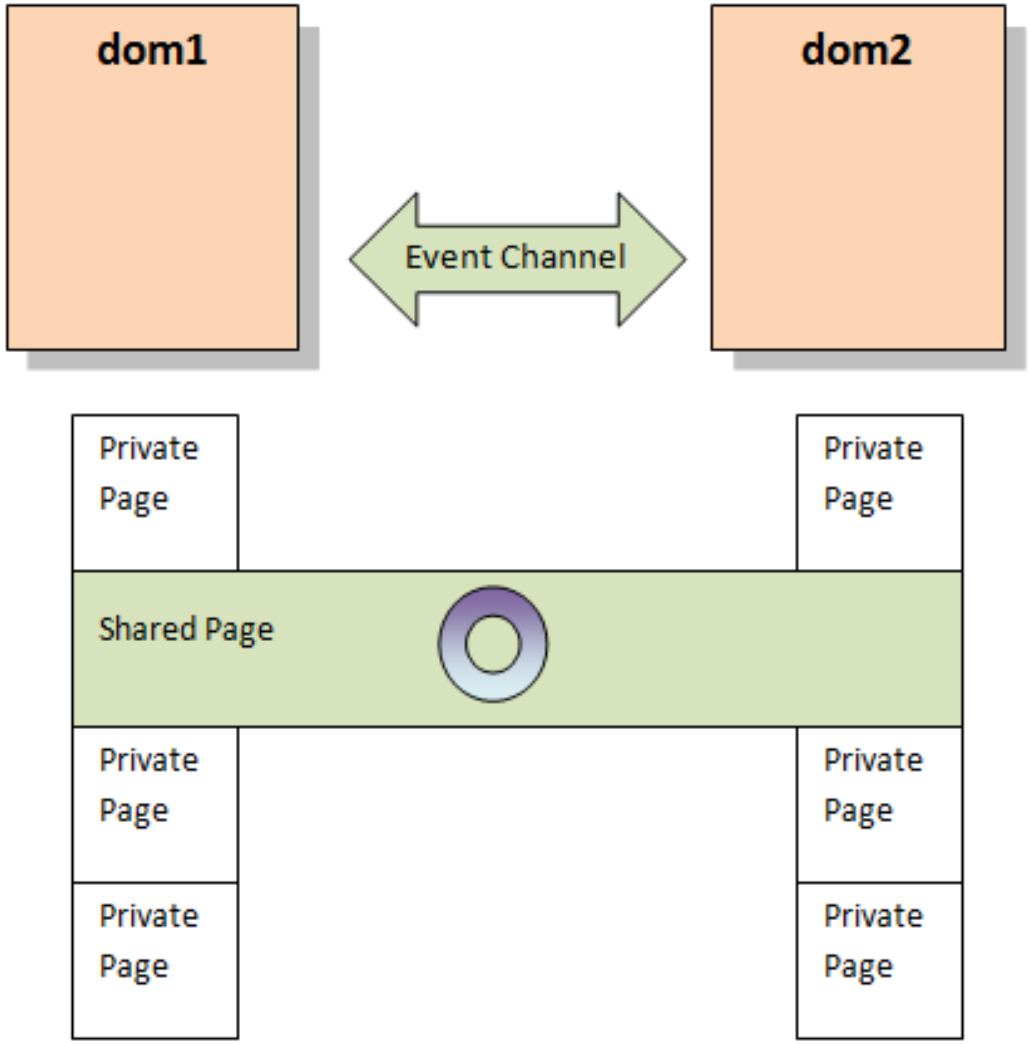
- ARINC 653 time and space partitioning
  - Refinement of Xen scheduling and memory management

- Dom 0
  - Relegated to System Configuration
  - Not Active during Normal mode operation



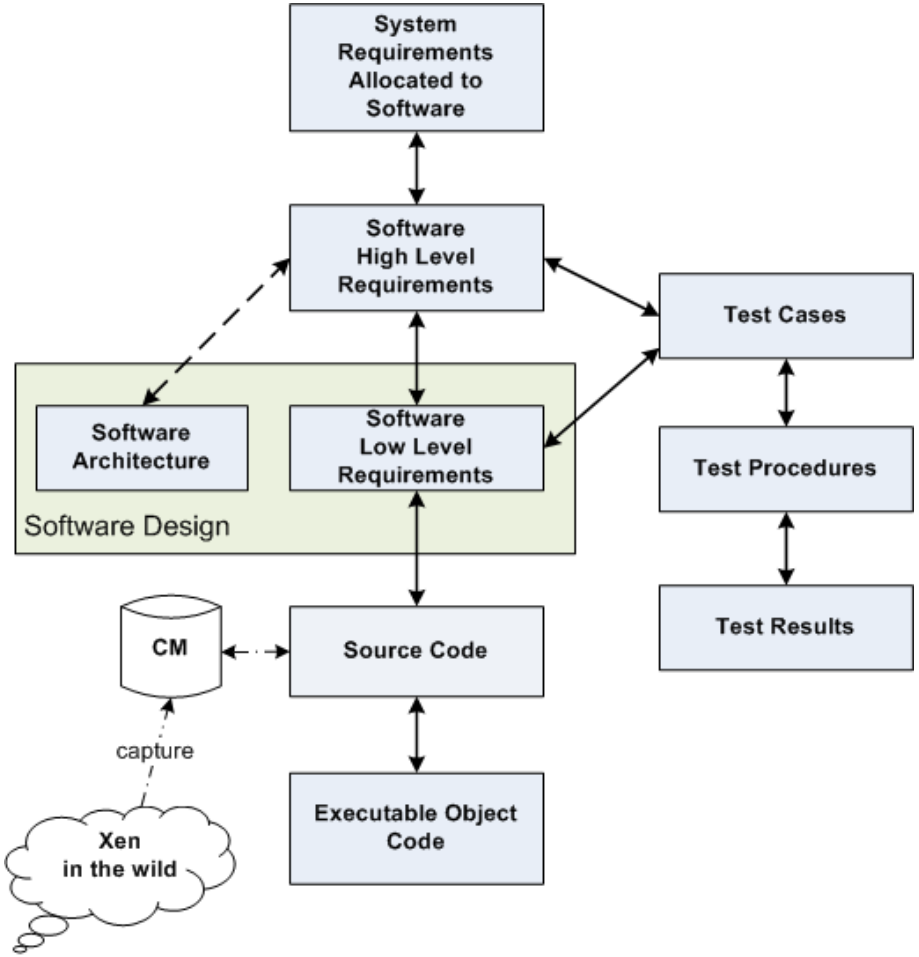
ARLX I/O and Inter-Partition Communication

- ARINC 653 Communication
  - Sampling Ports
  - Queuing Ports
- Privileged I/O DomU
  - In lieu of Dom0
    - Less Capable
  - Stripped down OS
- Enhanced I/O Manager
  - Refines 653 I/O Spec
  - Enforces Bandwidth Partitioning



Safety

- DO-178C Level A
  - Development Process
- ARLX Profile
  - Reduced SLOC
  - Reduced Complexity
- Reverse Engineer
  - Requirements
  - Architecture
  - Design Docs
  - V&V Tests



Security

- Common Criteria EAL6
  - Separation Kernel Protection Profile (SKPP)
    - High-Robustness Requirements

- Safety Properties
  - System does good things
  - **Shall** Requirements
    - Can be Tested
  - Well suited for DO-178C

- Security Properties
  - System doesn't do bad things
  - **Shall Not** Requirements
    - "Cannot" be tested
    - Can be Formally Verified
  - Addressed by Common Criteria

Formal Methods and the Common Criteria

- Common Criteria
  - Security Evaluation Framework
  - Ensures rigorous specification, implementation and evaluation of products against their requirements
  - Graduated Evaluation Assurance Levels
    - EAL1-EAL7
    - EAL5-7 Require “Formal Methods”

- Formal methods satisfies the following CC sections
  - ADV_SPM (Security Policy Modeling)
  - ADV_FSP (Functional Specification)
  - ADV_HLD (High-level Design)
  - ADV_LLD (Low-level Design)
  - ADV_RCR (Representation Correspondence)

Formal Methods

- Formal Methods
  - Discipline in which Mathematical Reasoning is Applied to the Development or Verification of Computer Systems
  - Formal Languages
    - Rigorously Defined Syntax and Semantics (Meaning)
  - Formal Tools
    - Computer Programs that Manipulate Formal Languages
    - Employ Logic and Rules of Inference
- Rigorous Specification and Analysis
  - Forces Attention to Easily Overlooked Details
- Ideally Part of Formal Process

$X < X + 1$

$(P \& Q) \Rightarrow P$

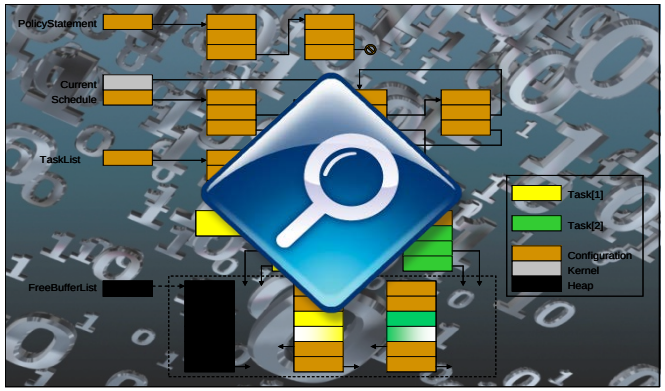
Recent Changes at NIAP

- The SKPP was sunset June 1, 2011
  - Part of a “new strategy”
  - Existing SKPP evaluations have revealed difficulties
    - maintenance, scalability, cost and complexity
  - No new SK protection profile is currently planned
- Content and Spirit of SKPP
  - Will continue to drive creation of assurance arguments and evidence
  - Foundation of security story
- Letter by Carol Saulsbury Houck, Director of NIAP
  - NIAP oversees US Common Criteria evaluations
  - A new paradigm is required
    - Expect to work with Security Community
      - Resolve Issues and Improve Results

The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the US Government

DFL: Data Flow Logic

- A Domain Specific Annotation Language
  - Information Flow Modeling & Analysis
  - Initial Work Sponsored by SPAWAR
    - LAW 2011



- Informed By
  - AAMP7, Green Hills
  - JML, SPARK

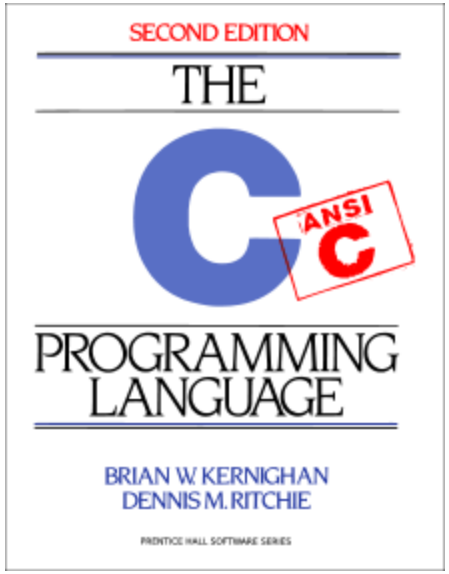
- Targeting C Source Code
  - Leverages GCC __attribute__ syntax
  - And C Macros

- Formal Foundation
  - Implemented in ACL2
  - Characterize as Semi-Formal (EAL6)

- Addresses Issues
  - Maintainability, Scalability, Evaluatability, Cost

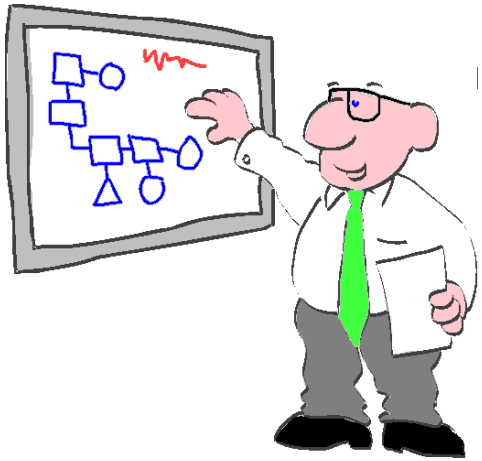
Specification Style

- DFL Specifications
  - Look a lot like C source programs
  - Leverage C declarations, preprocessor
- Supports Integrated Specifications
  - Specification is part of the source code
  - Compiler Compatibility
- Supports Federated Specifications
  - Specification is expressed outside of source code
  - Maintains Consistency



DFL Usage

- Raw (un-annotated) Analysis
  - Provides Early Flow feedback
  - Verbose
- Identify and Name Information (Security) Domains of Interest
  - DFL_DOMAIN TS,S,U;
- Articulate Flow Contracts Between Domains
  - DFL_FROM((TS,S,U))
- Classify System Variables w/to Domains
  - int key DFL_WITHIN((TS));
  - list DFL_CAST(Dlist) *ptr;
- Verify Contracts
  - Run Analyzer
  - Iterate



True Function

C Source Code

```
int test_sum2(int x, int y) {
  return(x + y);
}
```

DFL Report

```
procedure int test_sum2 (int x, int y)
 computes return()
     from x
     from y
```

Function with Globals

C Source Code

```
int a;

int global_sum (int x) {
  return(a+x);
}
```

DFL Report

```
procedure int global_sum (int x)
 computes return()
     from (global) a
     from x
```

Procedure with Side Effects

C Source Code

```
int a;

void global_write(int value) {
  a = value;
}
```

DFL Report

```
procedure void global_write (int value)
 computes return()
     from nothing
 computes (global) a
     from (global) a
     from value
```

Subtleties with Conditions

C Source Code

```
int test_if(int v1, int v2) {
  int res = 0;
  if (v1 == v2) {
    res = 1;
  }
  return(res);
}
```

DFL Report

```
procedure int test_if (int v1, int v2)
 computes return()
     from v1
     from v2
```

Common in Flow Analysis

Subtleties with Pointers

C Source Code

```
int int_from_star1(int *x) {
  return(*x);
}

int int_from_star2(int **x) {
  return(**x);
}
```

DFL Report

```
procedure int int_from_star1 ((int *) x)
 computes return()
     from x
     from (*x)

procedure int int_from_star2 (((int *) *) x)
 computes return()
     from x
     from (*x)
     from (*(*x))
```

Unique to C/DFL

Domains

- Domains
  - Primary Principals of High Level Policy
  - System Flow Policies Expressed w/to Domains
  - Program variables mapped into Domains

```
DFL_DOMAIN TS;

int key DFL_WITHIN((TS));
```

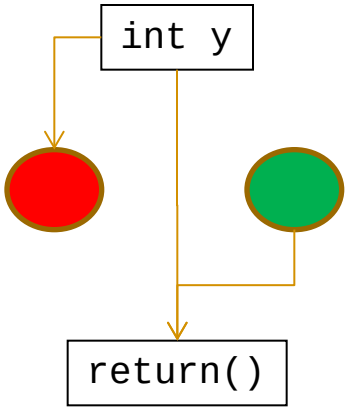
- DFL Domains enable collections of program variables and heap allocated data structures to be gathered together and treated collectively under a single name.
  - Variables A,B and C are “TOP SECRET”
  - Variables X,Y and Z are “UNCLASSIFIED”

- Giving such collections names allows us to articulate concise flow policies.
  - Variables inherit domain policies
  - “TOP SECRET” may get information from “UNCLASSIFIED”
    - “A,B and C” may get information from “X,Y and Z”

Policies

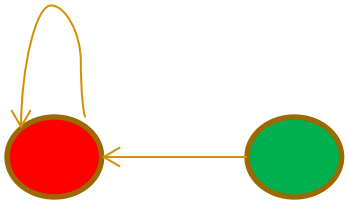
- Local Policies
  - Flows through Procedure Boundaries
    - Constrains intermediate flows
  - Definitions: used and checked
  - Declarations: used (assumed)

```
int foo(int y DFL_TO((TS)))
    DFL_FROM((U,y));
```



- Global Policies
  - Flows Between Domains
  - Policies inherited by Members
  - Always Checked

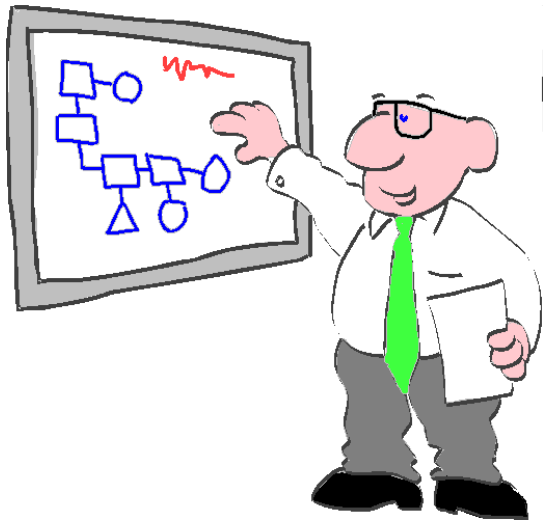
```
DFL_DOMAIN TS DFL_FROM((TS,U));
```



Classification

- Classification is the process of determining which objects to assign to which domains under what conditions.
  - Manual Process
  - Much of the Work

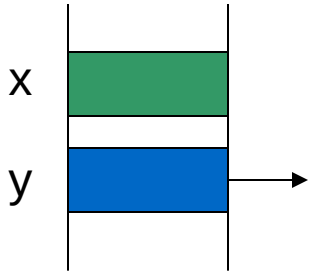
- The classification process
  - Maps A High Level Understanding
    - To the Low Level Implementation
  - Provides a context within which information flow policies may be accurately expressed and verified



Classifications

- Map Program Variables into Security Domains

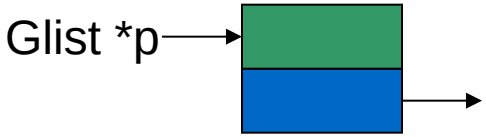
- Variable-based classifications classify named variables.
  - Global Variables
  - Procedure Arguments




```
int x DFL_WITHIN((U));

list * y DFL_WITHIN((C));

void foo(int x DFL_WITHIN((U)));
```

- Type-based annotations classify referenced objects

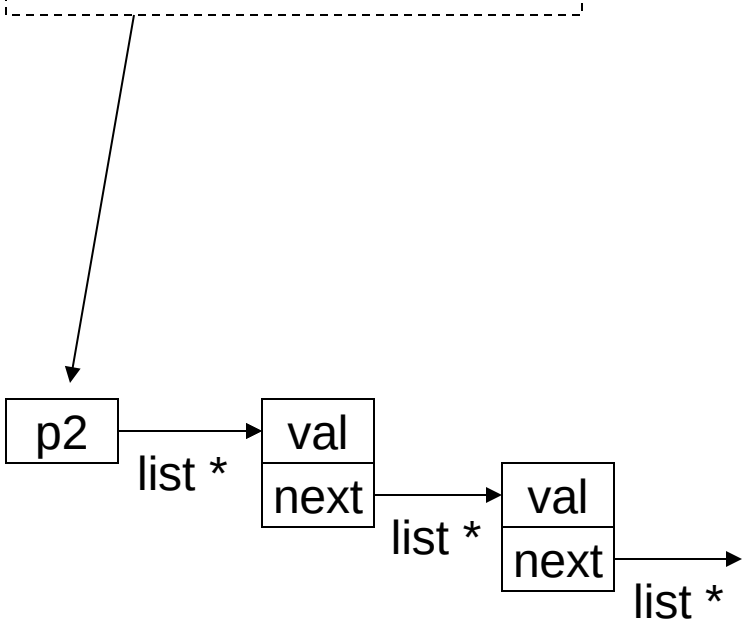
```
typedef struct Glist {
    int    val  DFL_WITHIN((U));
    Glist *next DFL_WITHIN((C));
} Glist;

list DFL_CAST(Glist) *p;
```



Interpreting Classifications

source.c

```
typedef struct list {
  int   val;
  list *next;
} list;

list *p2;
```



spec.h

```
#include "source.c"
DFL_DOMAIN TS,S,C,U;


typedef struct Glist {
  int    val  DFL_WITHIN((U));
  Glist *next DFL_WITHIN((U));
} Glist;





typedef struct Rlist {
  int    val  DFL_WITHIN((TS));
  Glist *next DFL_WITHIN((TS));
} Rlist;





list DFL_CAST(Rlist) *p2
     DFL_WITHIN((S))
     ;
```

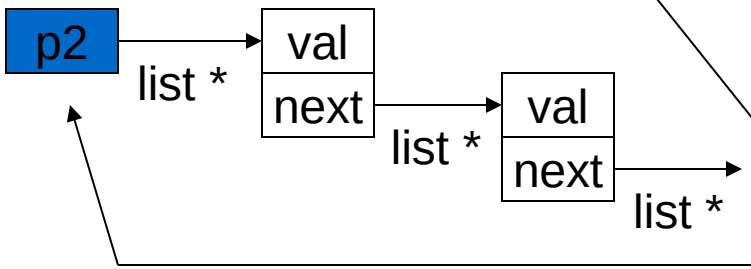
Interpreting Classifications

source.c

```
typedef struct list {
  int   val;
  list *next;
} list;

list *p2;
```

spec.h

```
#include "source.c"
DFL_DOMAIN TS,S,C,U;


typedef struct Glist {
  int    val  DFL_WITHIN((U));
  Glist *next DFL_WITHIN((U));
} Glist;




typedef struct Rlist {
  int    val  DFL_WITHIN((TS));
  Glist *next DFL_WITHIN((TS));
} Rlist;


list DFL_CAST(Rlist) *p2
     DFL_WITHIN((S))
     ;
```



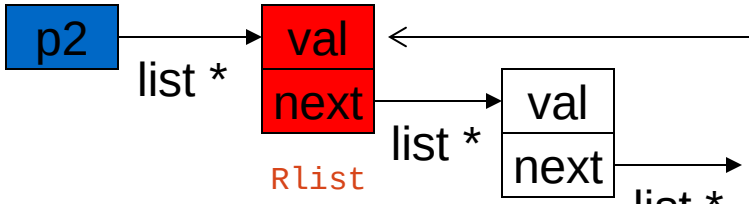
Interpreting Classifications

source.c

```
typedef struct list {
  int   val;
  list *next;
} list;

list *p2;
```

spec.h

```
#include "source.c"
DFL_DOMAIN TS,S,C,U;


typedef struct Glist {
  int    val  DFL_WITHIN((U));
  Glist *next DFL_WITHIN((U));
} Glist;


typedef struct Rlist {
  int    val  DFL_WITHIN((TS));
  Glist *next DFL_WITHIN((TS));
} Rlist;


list DFL_CAST(Rlist) *p2
     DFL_WITHIN((S))
     ;
```



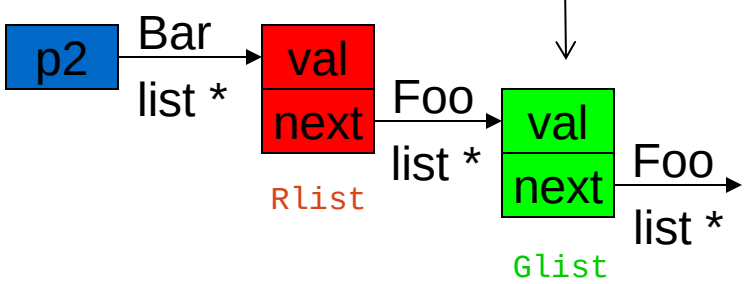
Interpreting Classifications

source.c

```
typedef struct list {
  int   val;
  list *next;
} list;

list *p2;
```

spec.h

```
#include "source.c"
DFL_DOMAIN TS,S,C,U;

typedef struct Glist {
  int    val  DFL_WITHIN((U));
  Glist *next DFL_WITHIN((U));
} Glist;

typedef struct Rlist {
  int    val  DFL_WITHIN((TS));
  Glist *next DFL_WITHIN((TS));
} Rlist;

list DFL_CAST(Rlist) *p2
     DFL_WITHIN((S))
     ;
```



Reasoning Example

source.c

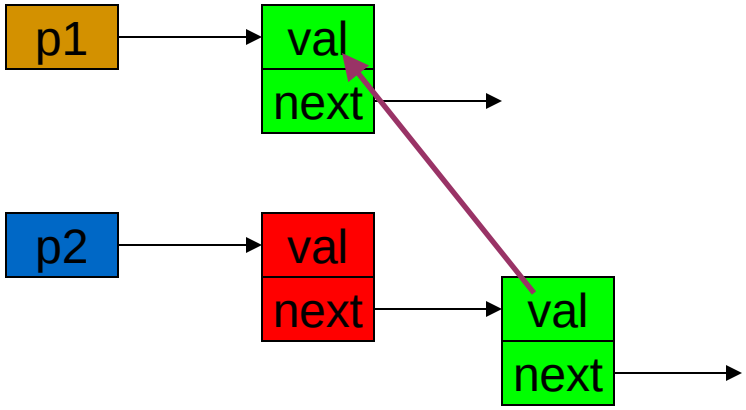
```
typedef struct list {
  int   val;
  list *next;
} list;

void BadBoy (list *p1, list *p2) {
  p1->val = p2->next->val;
}
```

spec.h

```
DFL_DOMAIN U  DFL_FROM((U));
DFL_DOMAIN TS DFL_FROM((TS,U));
DFL_DOMAIN S,C;

void BadBoy(
      list DFL_CAST(Glist) *p1
           DFL_WITHIN((C)),
      list DFL_CAST(Rlist) *p2
           DFL_WITHIN((S))
     )
  DFL_CALL_TO((U));
```

```
U  <- U
TS <- TS + U
```

```
U <- U
```



Reasoning Example

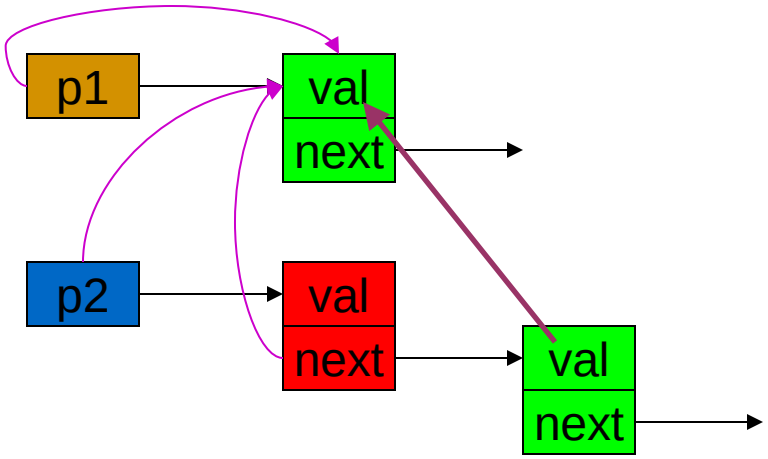
source.c

```
typedef struct list {
  int   val;
  list *next;
} list;

void BadBoy (list *p1, list *p2) {
  p1->val = p2->next->val;
}
```

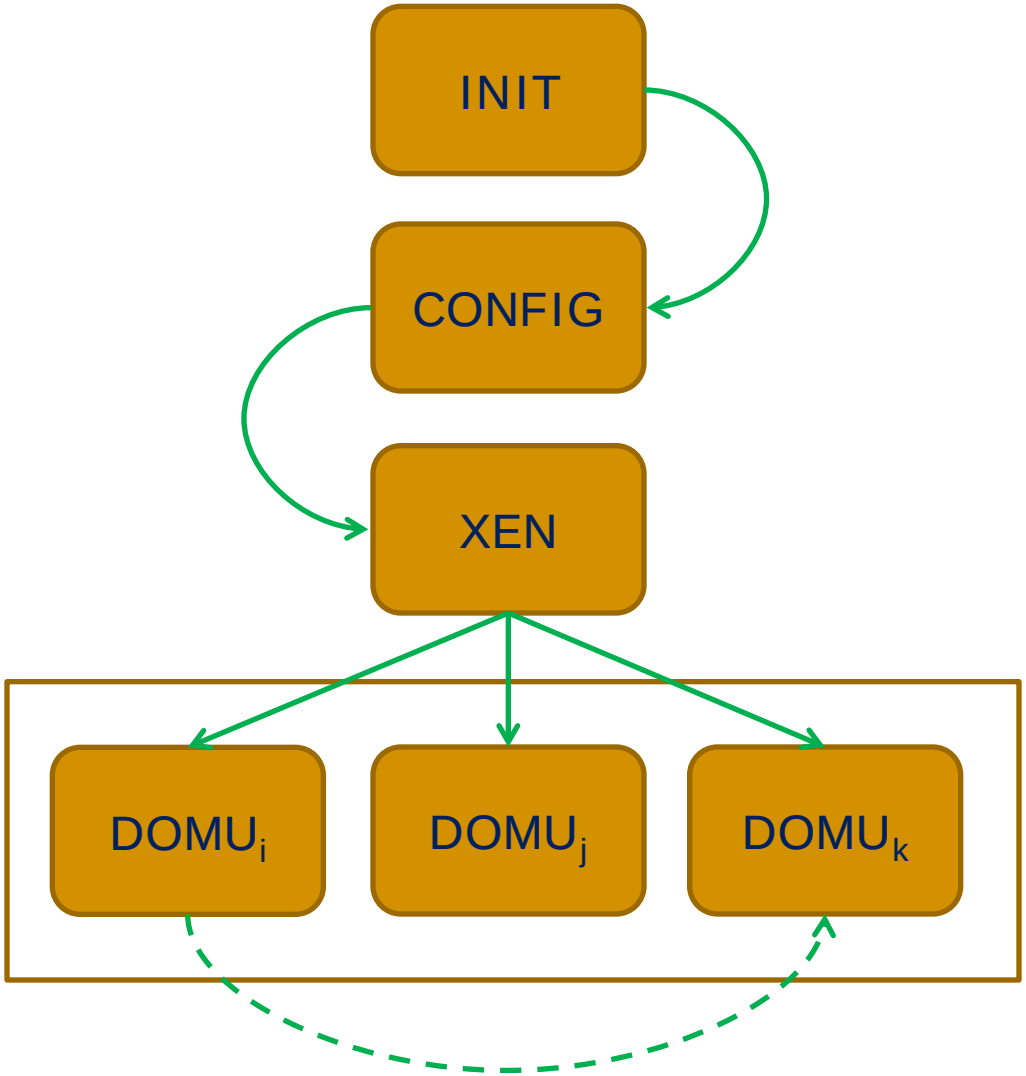
exceptions report

```
procedure BadBoy (Glist * p1,
                  Rlist * p2)
 computes (* p1).val
   within U
     from p1 within C
     from p2 within S
     from (* p2).next within TS
```

```
U <- C + S + TS + U
```



ARLX Information Flow Security Policy



ARLX Global Classifications

- Global Variables
  - Inputs, Outputs, Channels
- ARLX TOE
  - 195 Global Variables

```
time_t next_switch_time
       DFL_WITHIN((ARLX_XEN));

int sched_index
    DFL_WITHIN((ARLX_XEN));

struct vcpu *idle_vcpu[56]
    DFL_WITHIN((ARLX_XEN));
```

ARLX Type Classifications

- Heap Objects
  - Store System State
  - Must be Classified
- Types
  - Means of Access
- ARLX TOE
  - 138 unique structures
  - Many fields

```
struct a653sched_priv_s {
  …
  s_time_t major_frame
    DFL_WITHIN((ARLX_CONFIG)) ;
  s_time_t next_major_frame
    DFL_WITHIN((ARLX_XEN)) ;
  struct list_head vcpu_list
    DFL_WITHIN((ARLX_CONFIG)) ;
  …
}
```

Interface (API) Contracts

- Declared but Undefined Procedures
  - Must still be characterized
- Contracts are used
  - But not verified
- Analysis Axioms
  - Must be Validated Manually
- ARLX TOE
  - 53 Interfaces

```
long __builtin_expect() DFL_CONST ;

int vcpu_runnable(struct vcpu *v)
  DFL_PURE
  DFL_FROM((v,ARLX_XEN))
  ;
```

Procedure Contracts

- Every Procedure
  - Must have a contract
- ARLX TOE
  - 551 procedures
  - 26K SLOC
- Contracts are Verified
  - OK to mess up
- Default Contracts
  - Auto Generated
  - May need refinement (iterative)

```
struct task_slice
    a653sched_do_schedule(
        struct scheduler *ops
            DFL_WITHIN((ARLX_XEN)) ,
        s_time_t now
            DFL_WITHIN((ARLX_XEN)) ,
        bool_t tasklet_work_scheduled
            DFL_WITHIN((ARLX_XEN))
        )
  DFL_CALL_TO((ARLX_XEN))
  ;
```

Function Pointer Contracts

- Indirect Calls
  - Function Pointers
  - Quite Common
- Similar to APIs
  - Don't know what is behind the call
  - Many are initialized
- Contracts
  - Via Type annotations
  - Constrain behavior
- ARLX TOE
  - 30 procedures

```
struct scheduler {
  ...
  struct task_slice (*do_schedule)
     (struct scheduler *
        DFL_WITHIN((ARLX_XEN)) ,
      s_time_t
        DFL_WITHIN((ARLX_XEN)) ,
      bool_t
        DFL_WITHIN((ARLX_XEN)) )
    DFL_CALL_TO((ARLX_XEN)) ;
  ...
}
```

Assembly Code

- Unsupported
  - Renders Analysis Unsound
  - Requires Manual Evaluation
  - Reported by DFL as a call to _ASM_
- ARLX TOE
  - 178 procedures infected
  - Introduced via macros

```
unsigned int __scanbit(unsigned long val , unsigned long max ) {

 __asm__("bsf %1,%0 ; cmovz %2,%0": "=&r" (val): "r" (val), "r" (max));

  return ((unsigned int )val);

}
```

Initial Analysis Results

- Analyzed Schedule TOE
  - a653_do_schedule()
  - 26K SLOC
  - 15 min
- Policy Exceptions
  - Assembly Code
  - Pointer Arithmetic
- Minimize Exceptions
  - Exceptions must be manually evaluated

```
struct task_slice
  a653sched_do_schedule(
    struct scheduler *ops,
    s_time_t now,
    bool_t tasklet_work_scheduled
  )
  computes return()
    from ((local) ret)
  calls _ASM_
```

Conclusion

- Objectives of Evaluation
  - Model and Analyze ARLX Scheduling Subsystem (TOE)
    - a653_do_schedule()
  - Establish Viability of DFL Approach
  - Identify Residual Issues
- Modeled Scheduling Subsystem
  - DFL Annotations, Results usable by Developers
- Analyzed Xen Code base
  - 26K SLOC, 15min
- Residual Issues
  - ARLX
    - Assembly Code
    - Pointer Arithmetic
  - DFL
    - Generic Policies
    - Automate Contract Generation