Towards proving security in the presence of large untrusted components

June Andronick
David Greenaway
Kevin Elphinstone

Australian Government
Department of Communications, Information Technology and the Arts
Australian Research Council

NICTA Members

NICTA Partners

Computers and Trust

Computers and Trust

Computers and Trust

Computers and Trust

Computers and Trust

Computers and Trust

- Advances in formal methods techniques give us hope
- The seL4 microkernel is one such example: around 10 thousand lines of code formally proven
  - approximately 25 person years of effort

Computers and Trust

- Advances in formal methods techniques give us hope
- The seL4 microkernel is one such example: around 10 thousand lines of code formally proven
  - approximately 25 person years of effort
- A typical smartphone consists of over 10 million lines of code

Computers and Trust

- Advances in formal methods techniques give us hope
- The seL4 microkernel is one such example: around 10 thousand lines of code formally proven
  - approximately 25 person years of effort
- A typical smartphone consists of over 10 million lines of code

Computers and Trust

- Advances in formal methods techniques give us hope
- The seL4 microkernel is one such example: around 10 thousand lines of code formally proven
  - approximately 25 person years of effort
- A typical smartphone consists of over 10 million lines of code

How can we provide *any* formal assurance to real-world systems of such size?

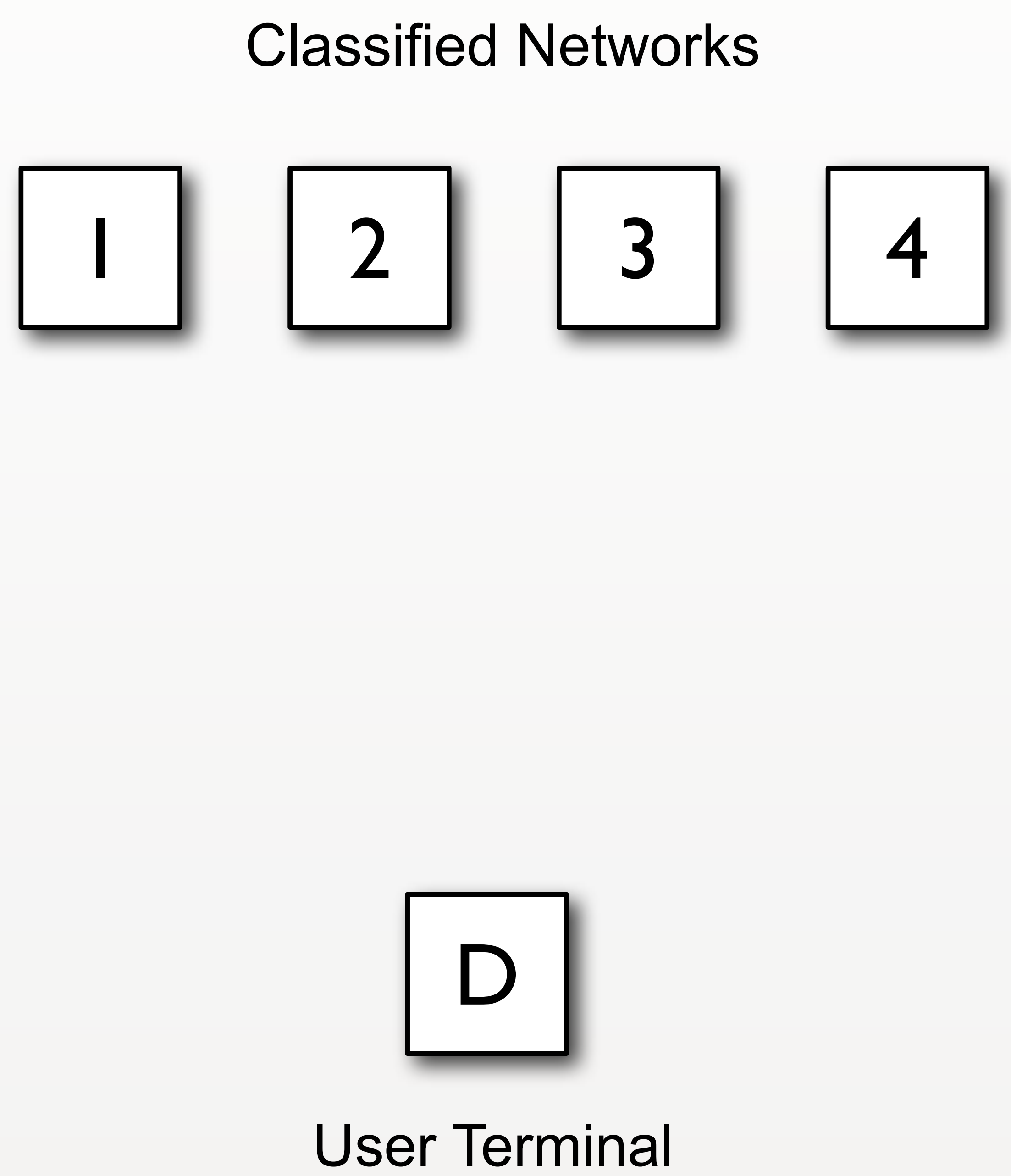
Our Vision

- Provide full system guarantees for *targeted* properties

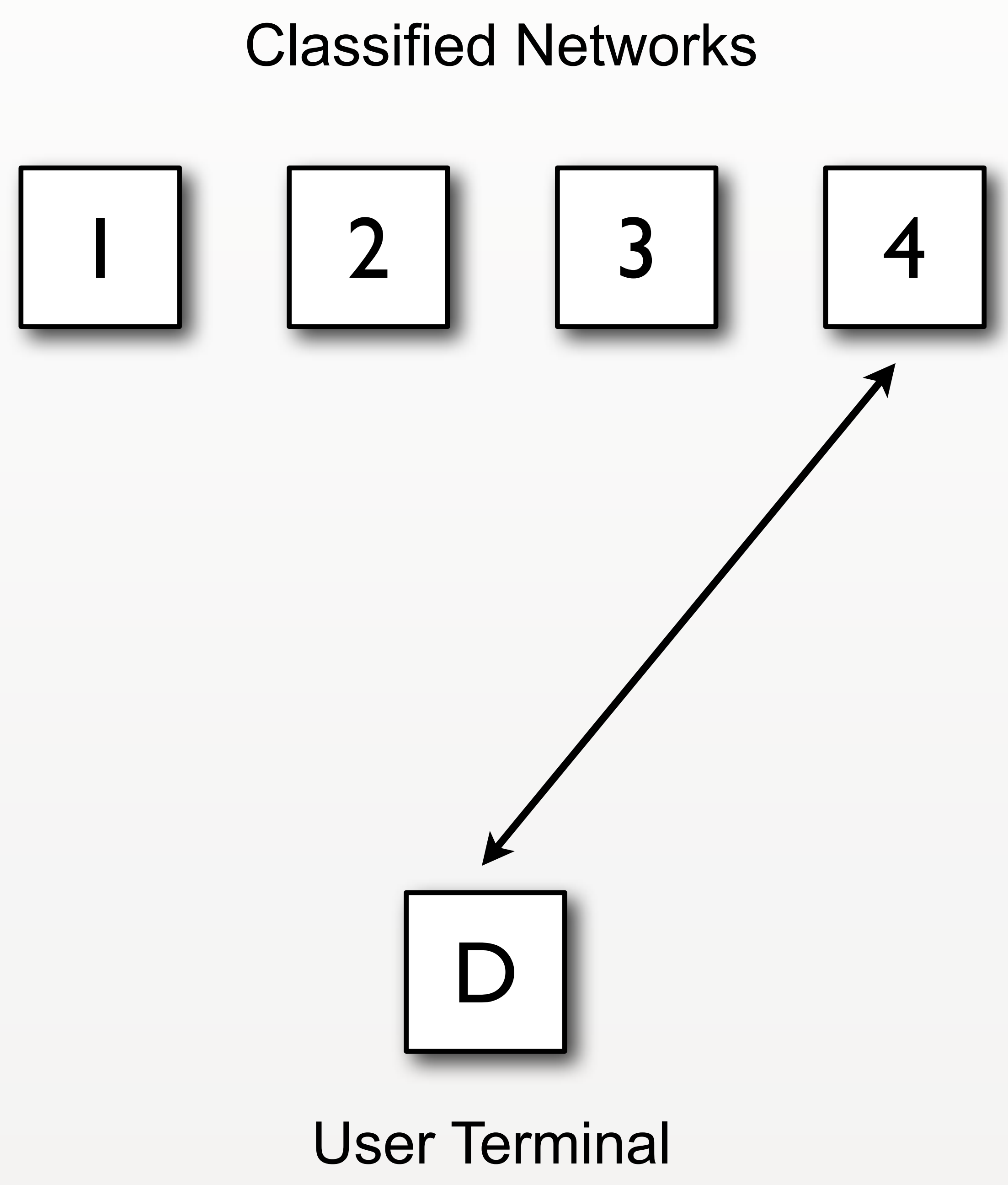
- Isolate the software parts that are not critical to the target property
  - And then prove that nothing more needs to be said about it

- Formally prove that the remaining parts satisfy the target property

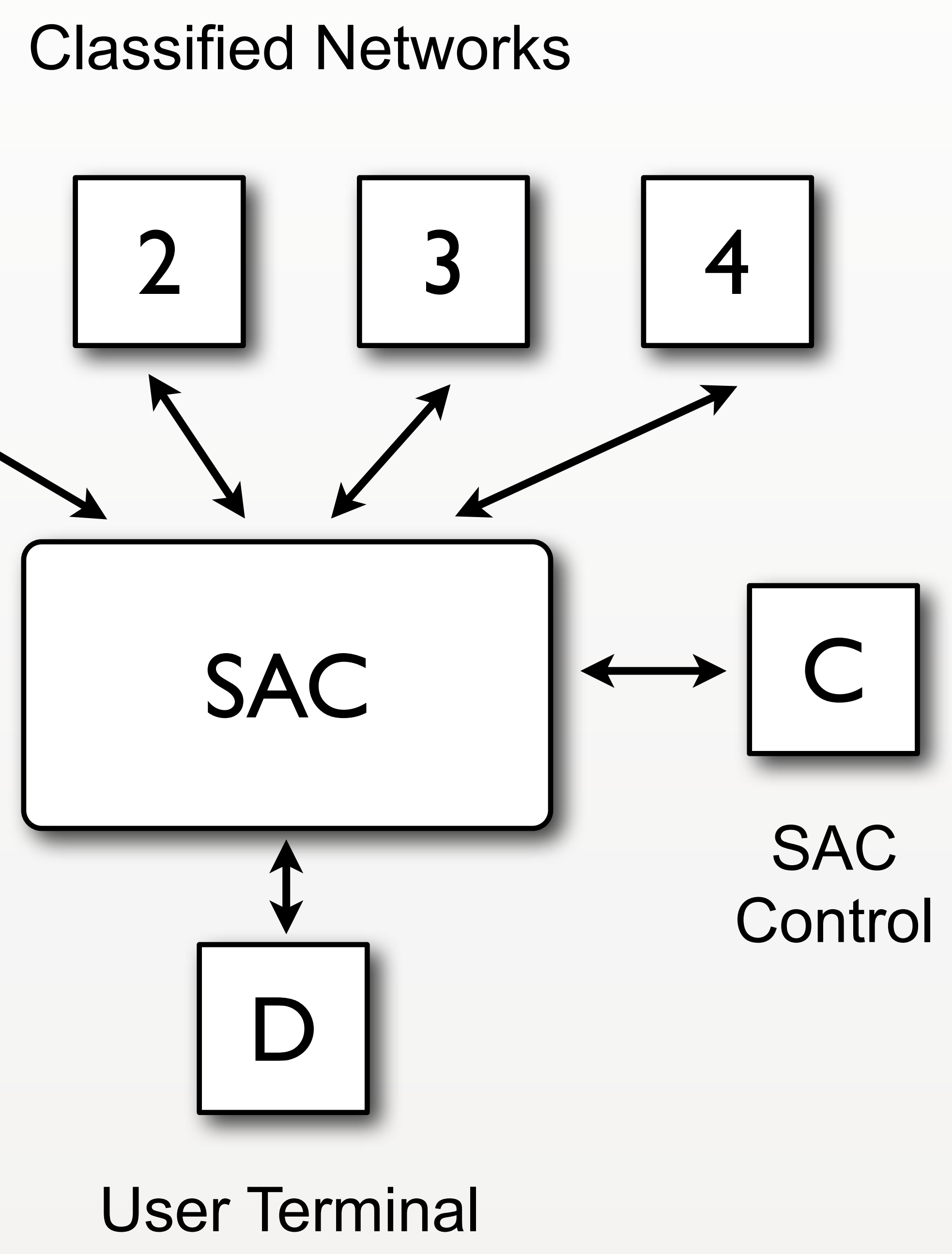
Case Study: Secure Access Controller



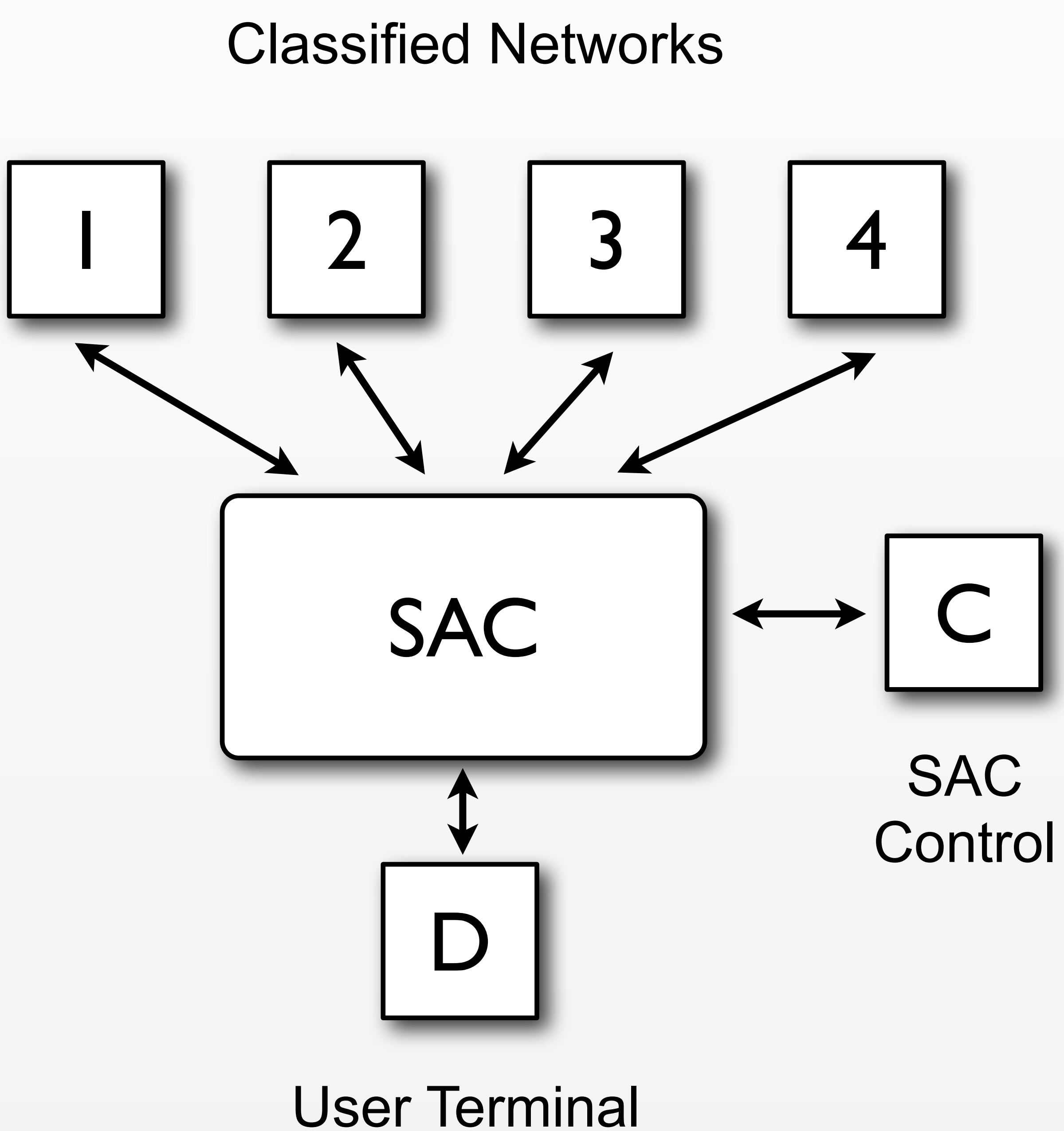
Case Study: Secure Access Controller



Case Study: Secure Access Controller



Case Study: Secure Access Controller



- Data from one classified network must not reach another
- Assumptions:
  - User terminal will not leak data
  - Only verify overt communication channels
  - All networks are otherwise malicious

Case Study: Secure Access Controller

SAC

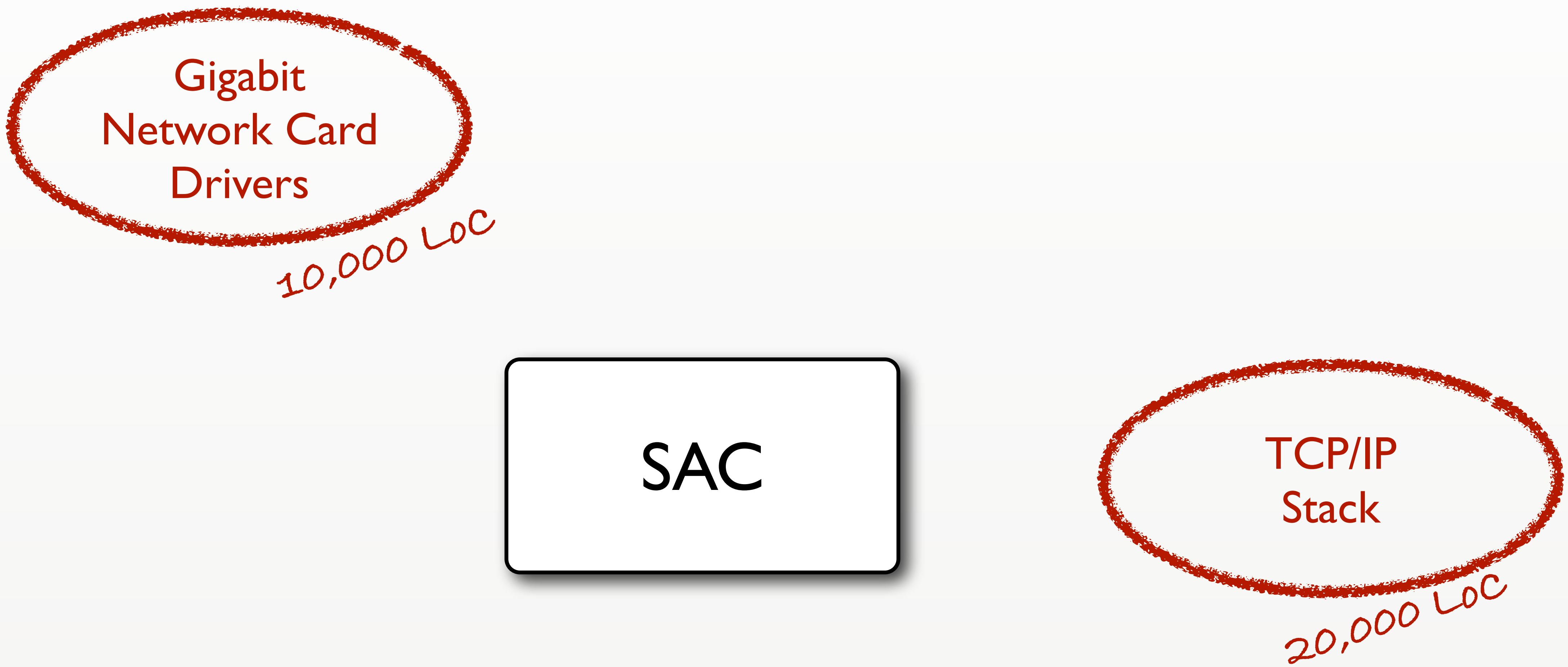
Case Study: Secure Access Controller

Gigabit
Network Card
Drivers

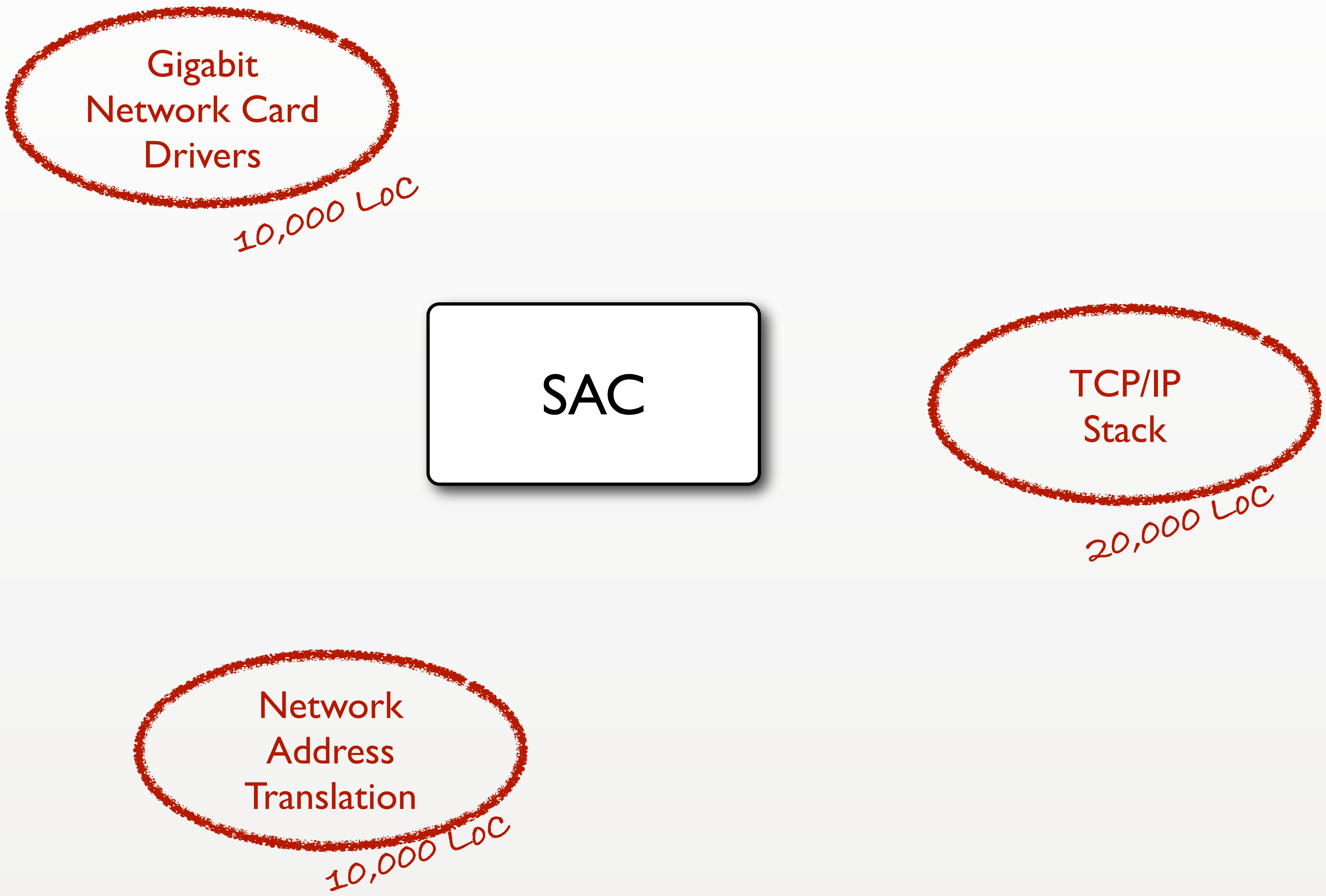
10,000 LoC

SAC

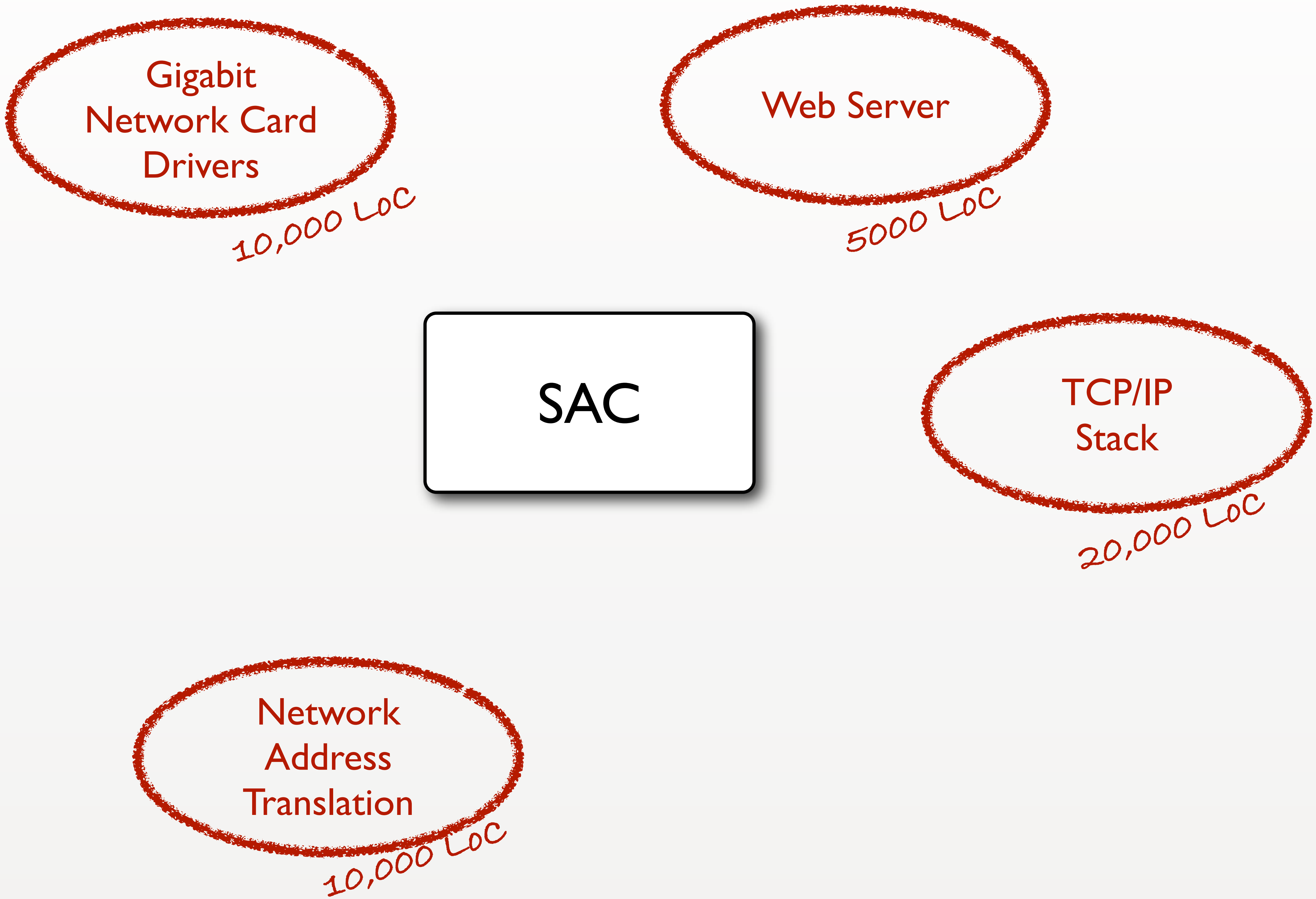
Case Study: Secure Access Controller



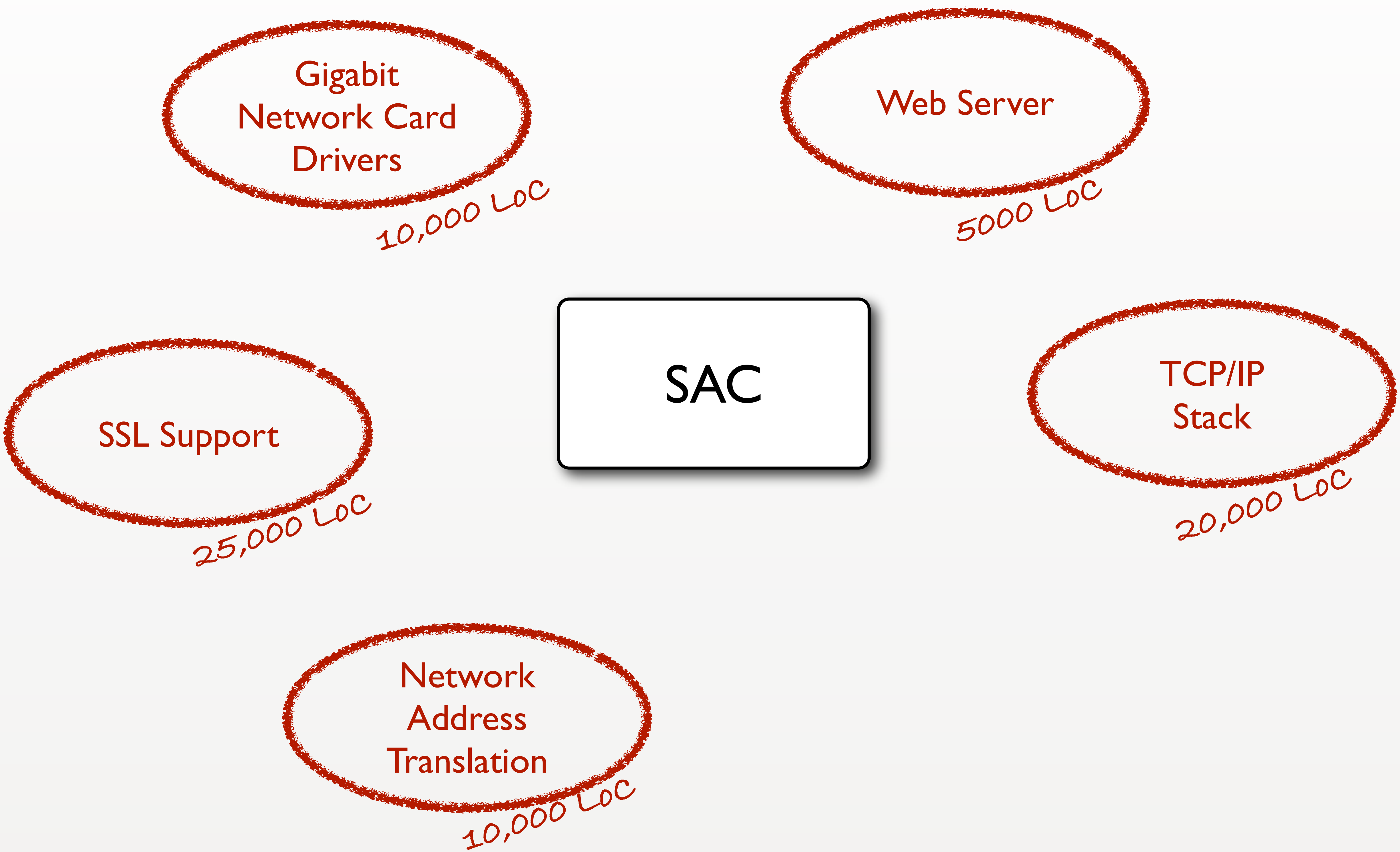
Case Study: Secure Access Controller



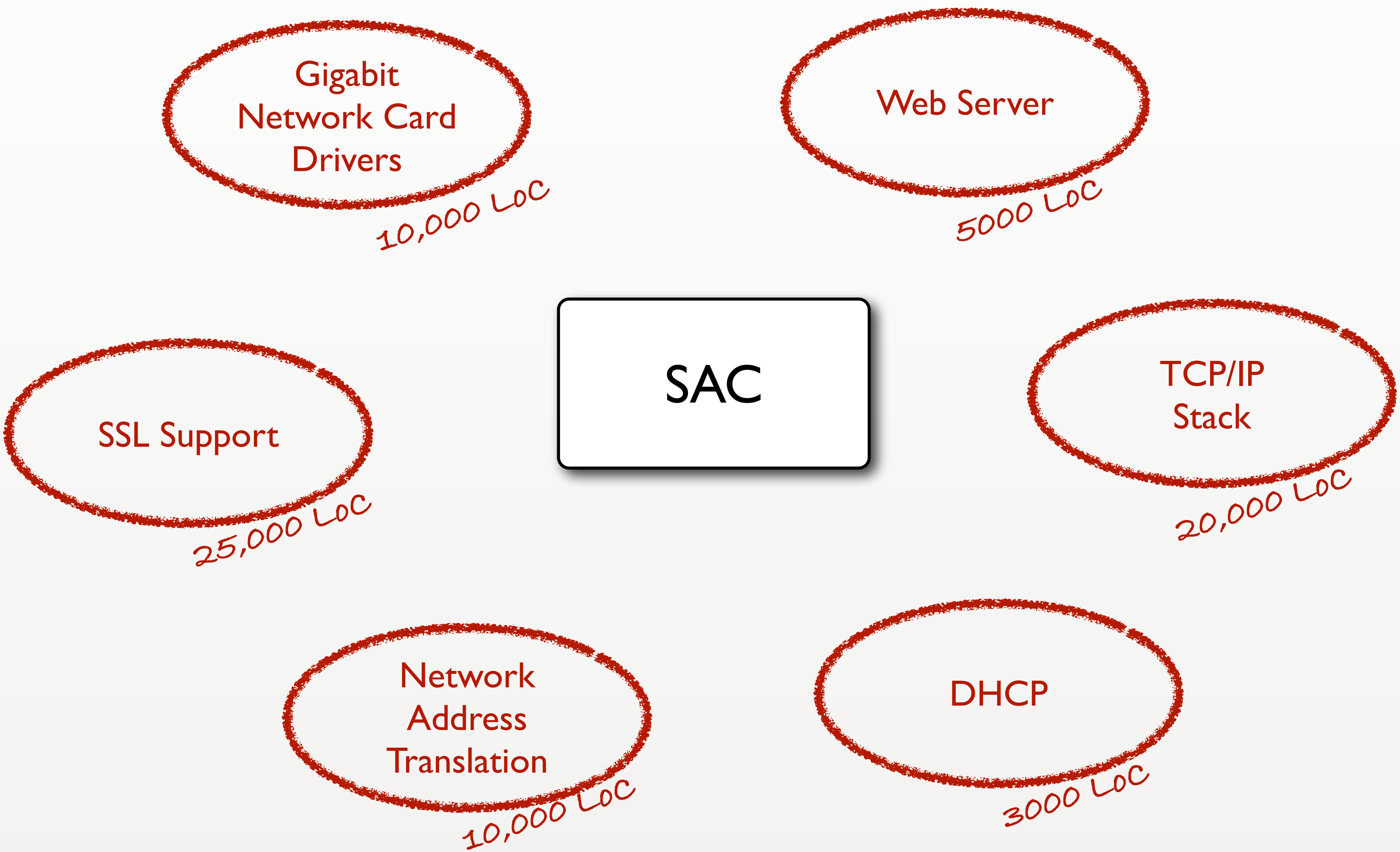
Case Study: Secure Access Controller



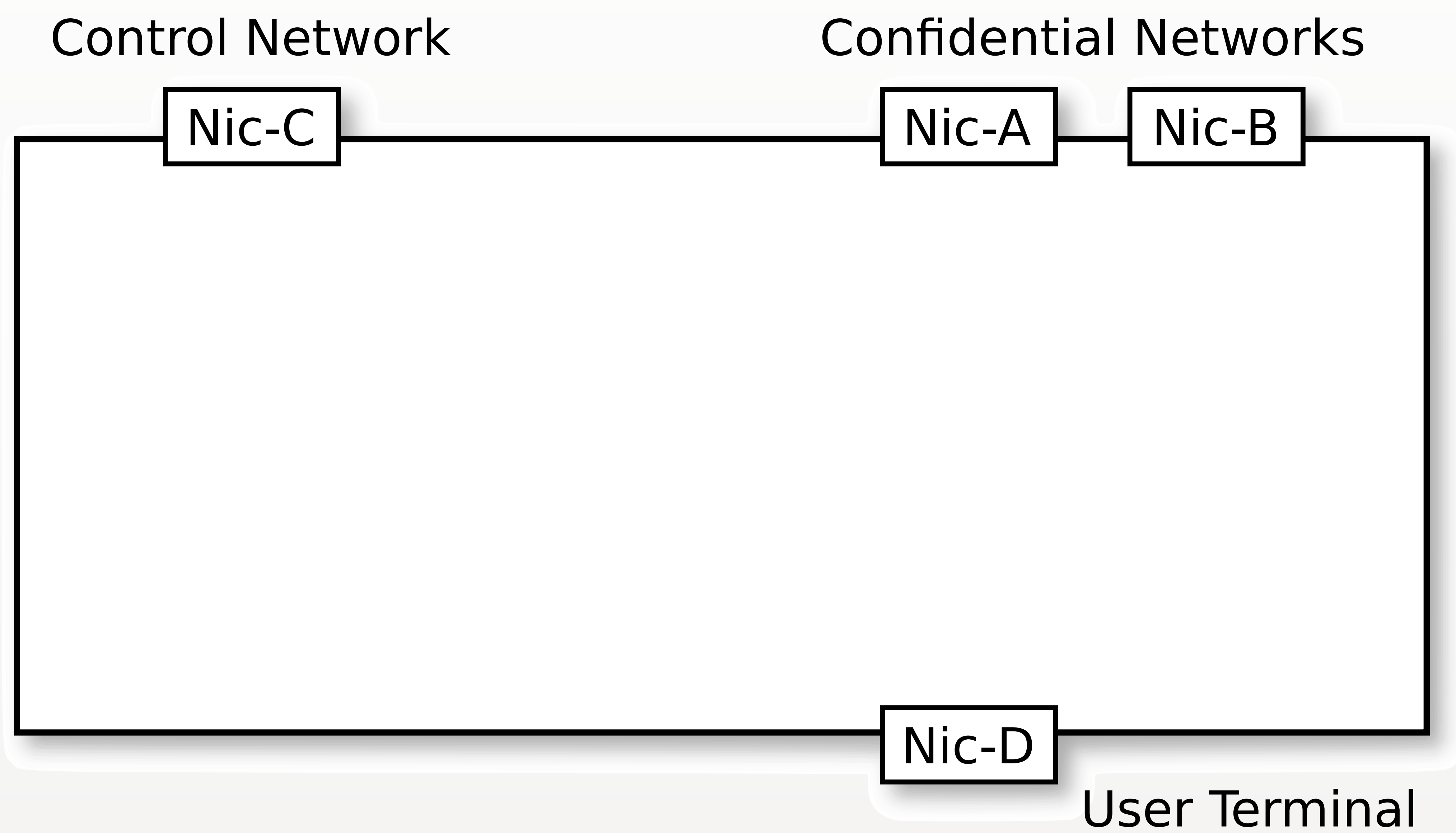
Case Study: Secure Access Controller



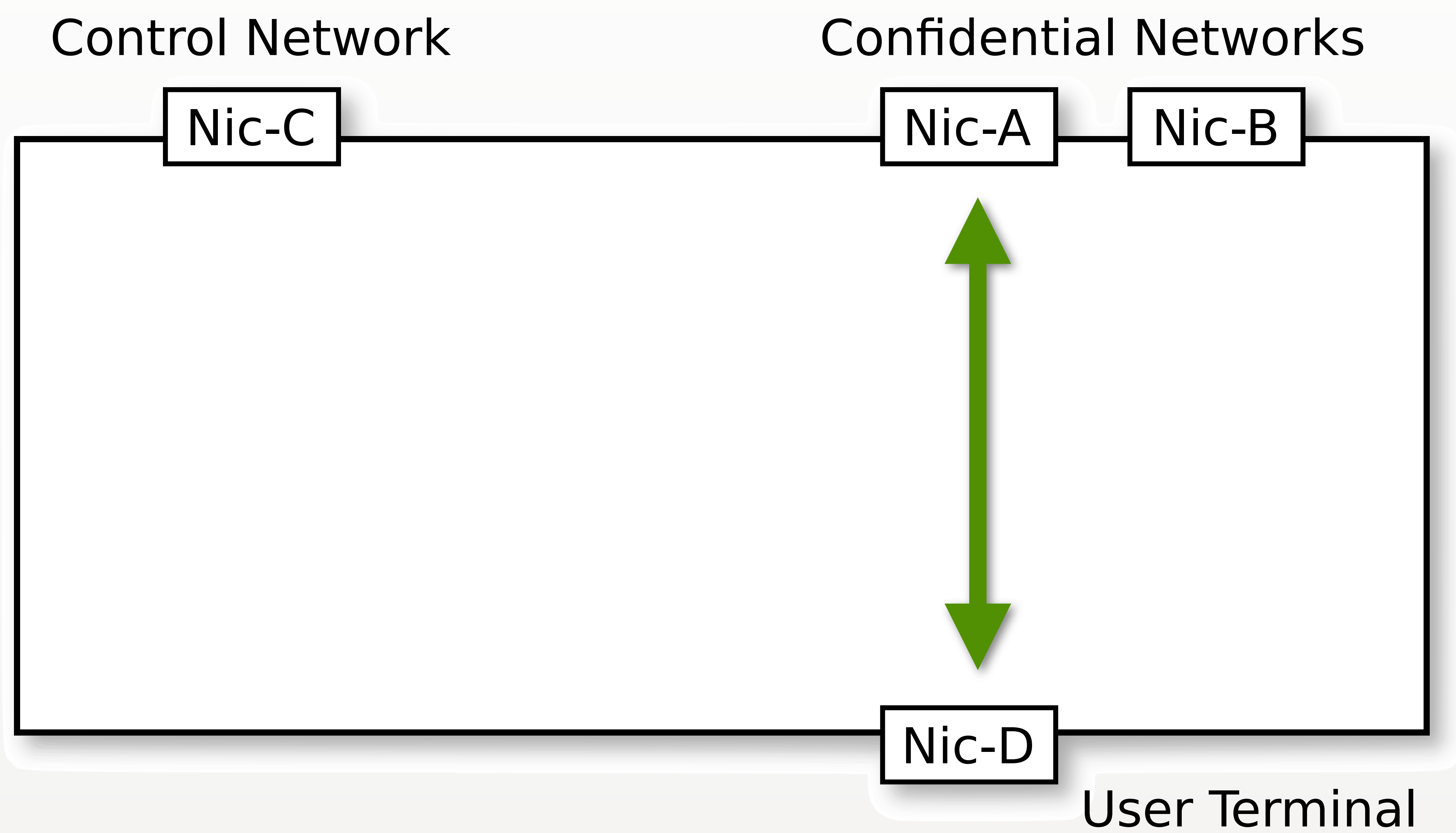
Case Study: Secure Access Controller



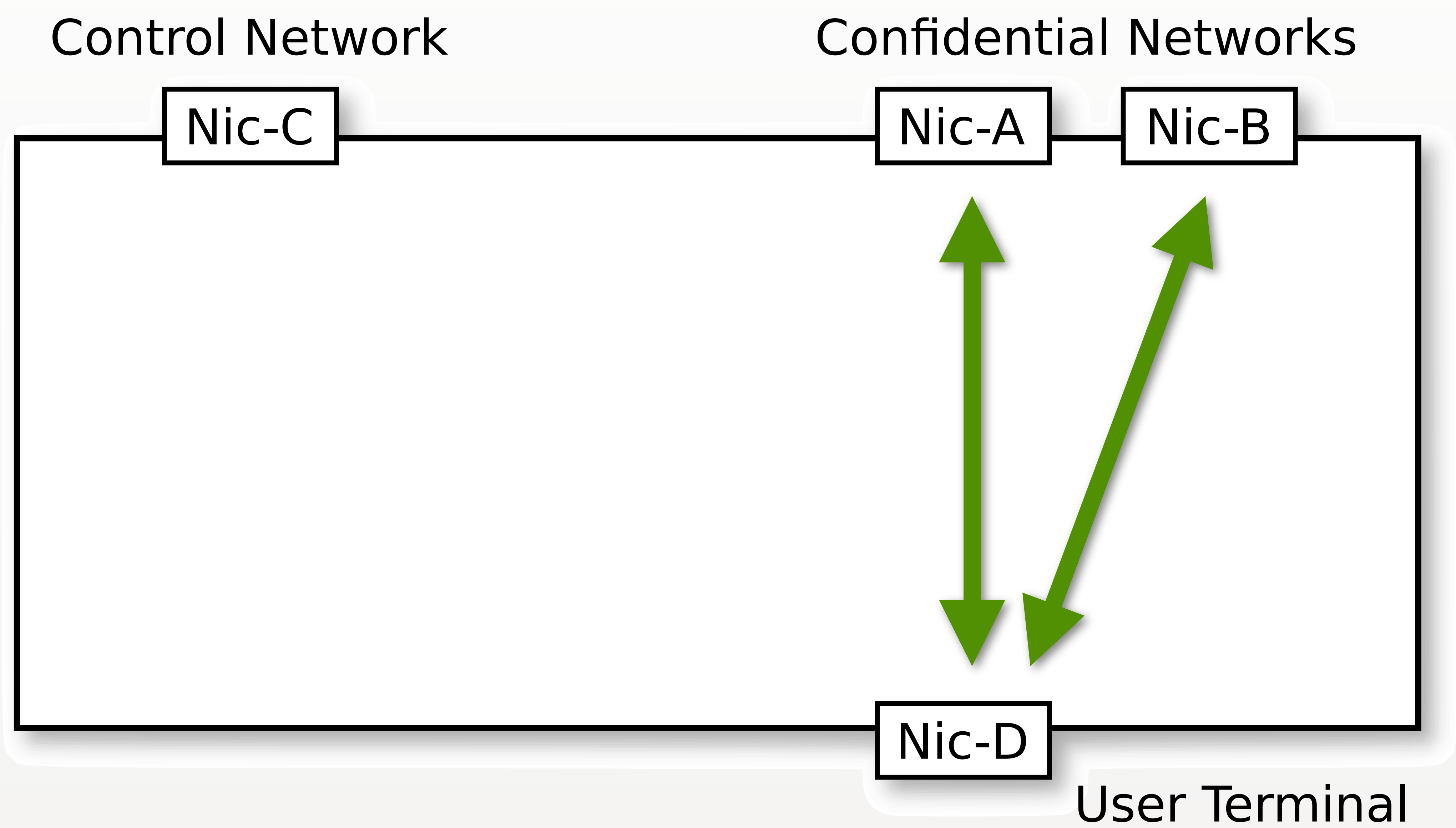
Case Study: Secure Access Controller



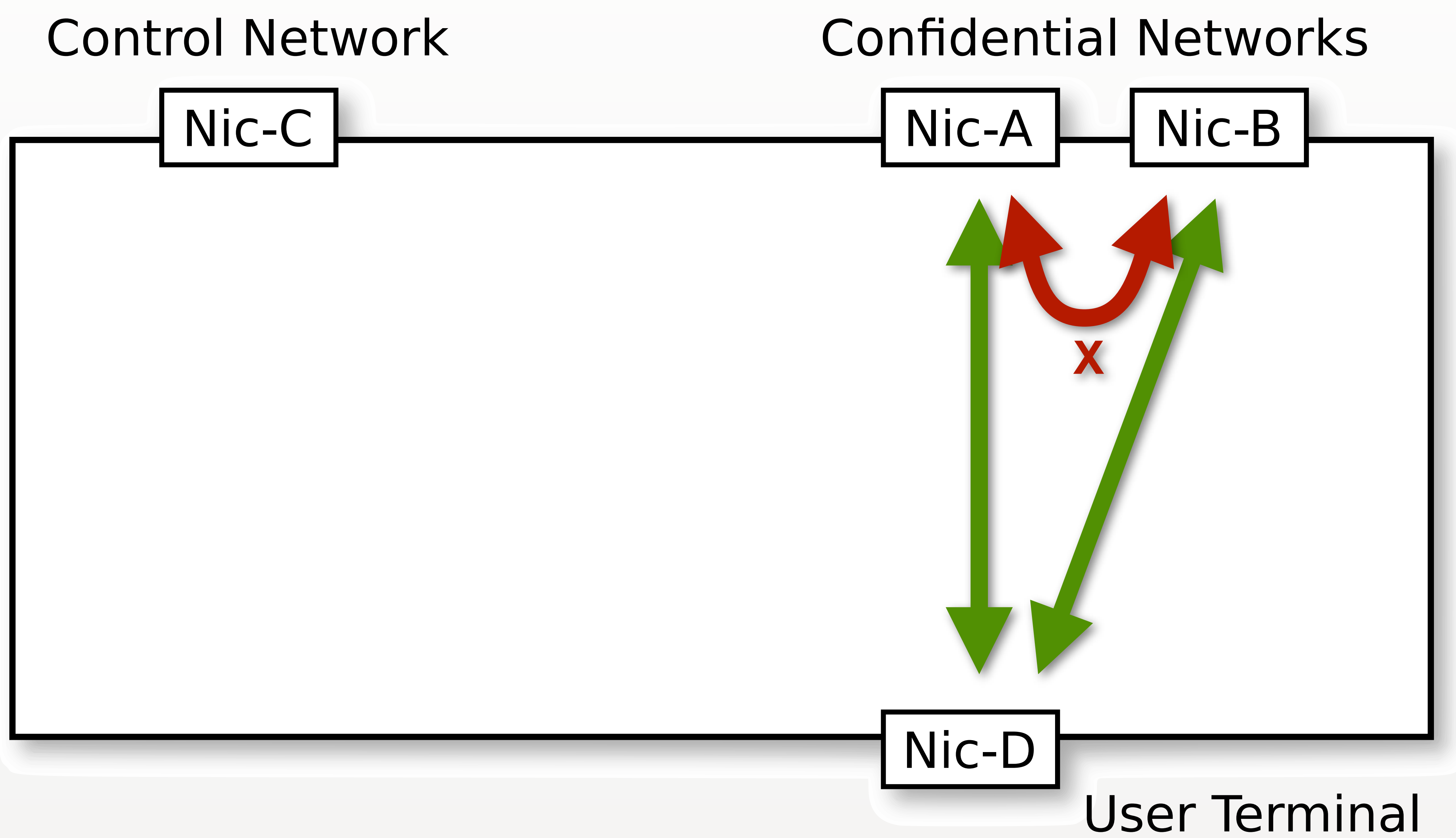
Case Study: Secure Access Controller



Case Study: Secure Access Controller



Case Study: Secure Access Controller



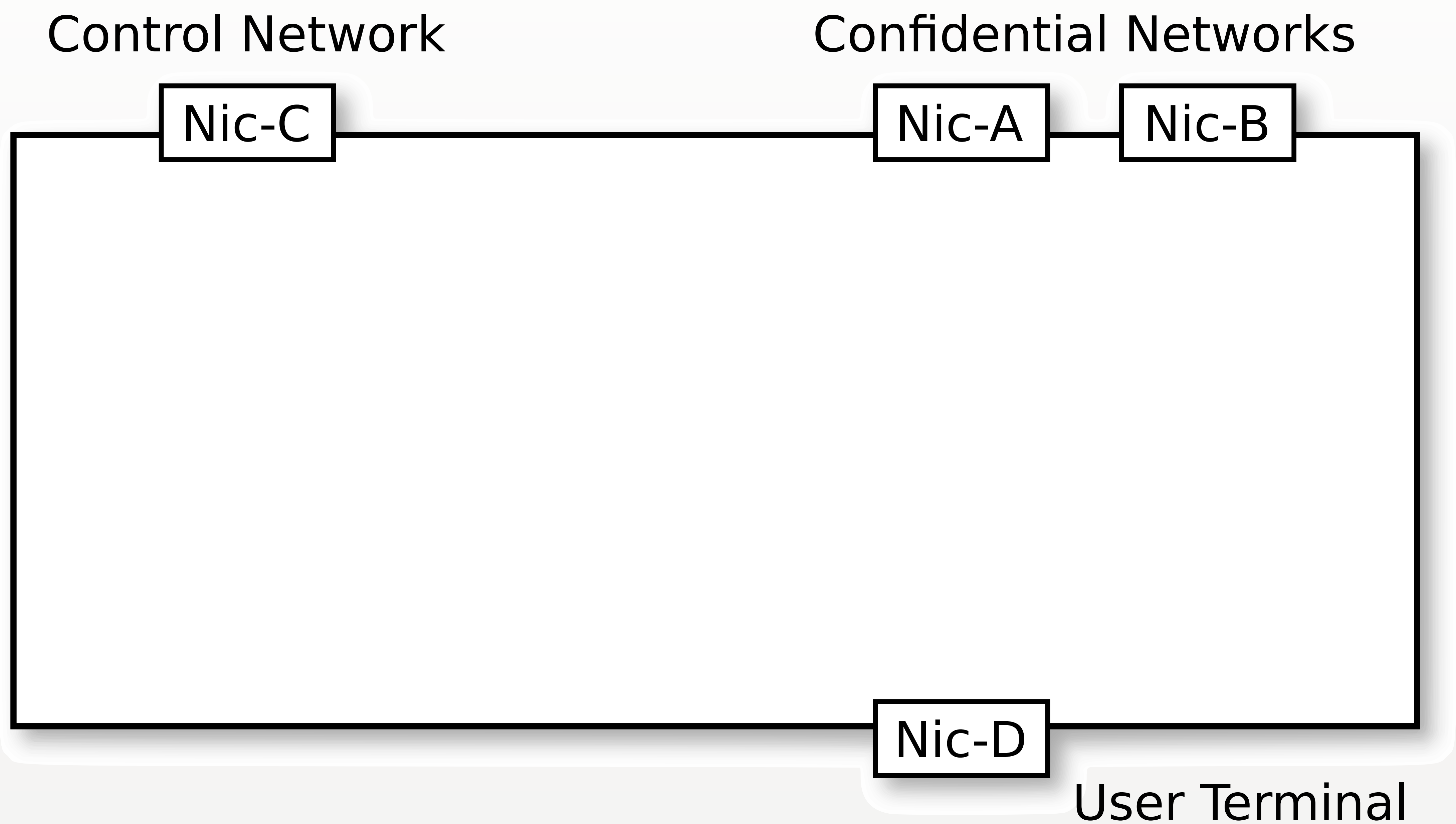
Case Study: Secure Access Controller

- Verification of all code in the system is infeasible
- Instead, split up code into components
  - Trusted / untrusted components
  - Only give components access to resources they need
  - Principle of least privilege
- To do this, we need some mechanism to enforce this split

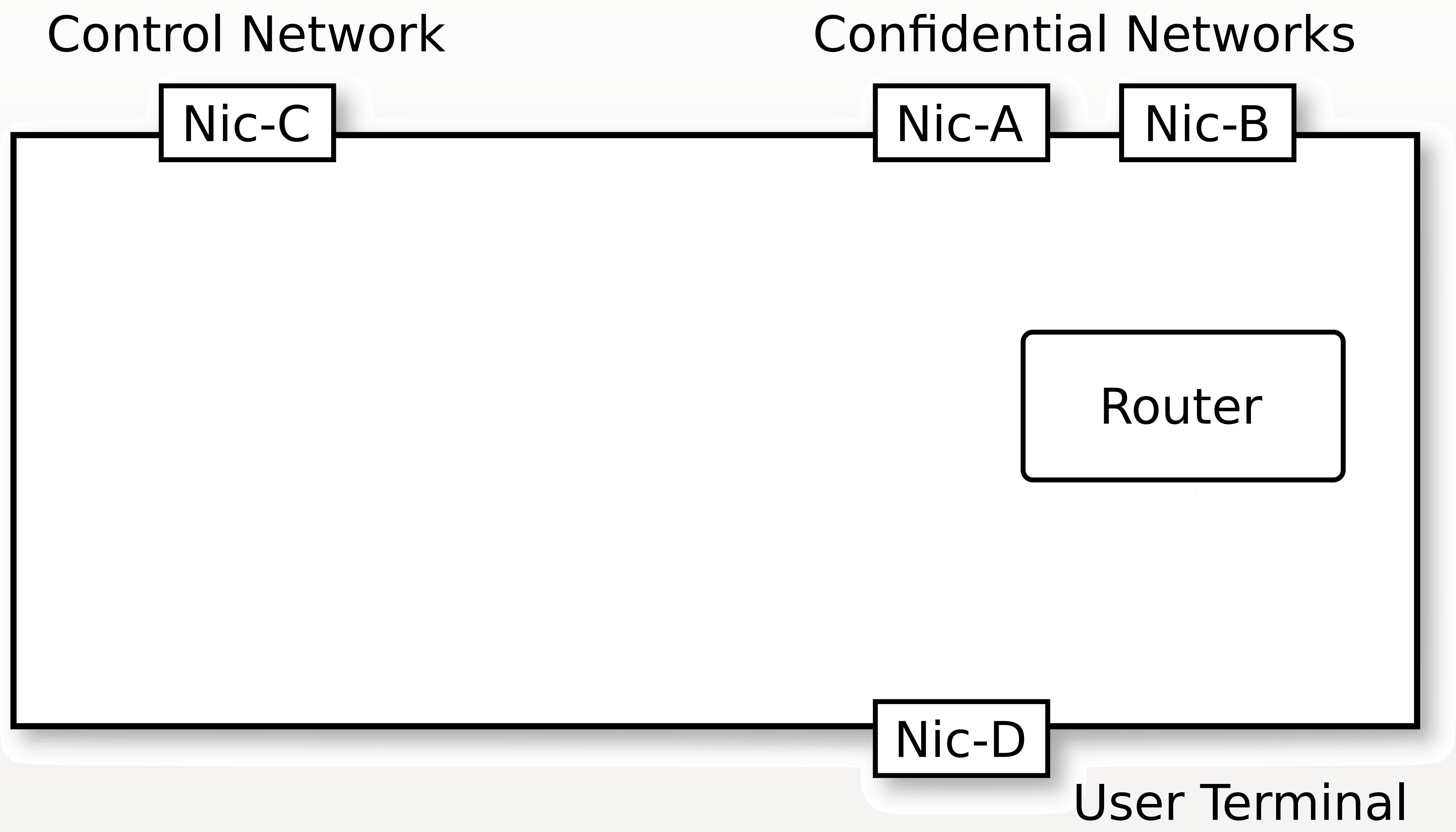
seL4 Microkernel

- Small operating system kernel
  - Threads
  - Address Spaces
  - Communication primitives
- Capability based
  - All system resources require a cap to be accessed
  - Provides access control, allowing threads to be isolated by using an appropriate cap distribution
- Proven functionally correct
  - seL4's C code shown to correctly implement its specification
    - Assumes correctness of hardware, compiler, initialisation code, assembly paths

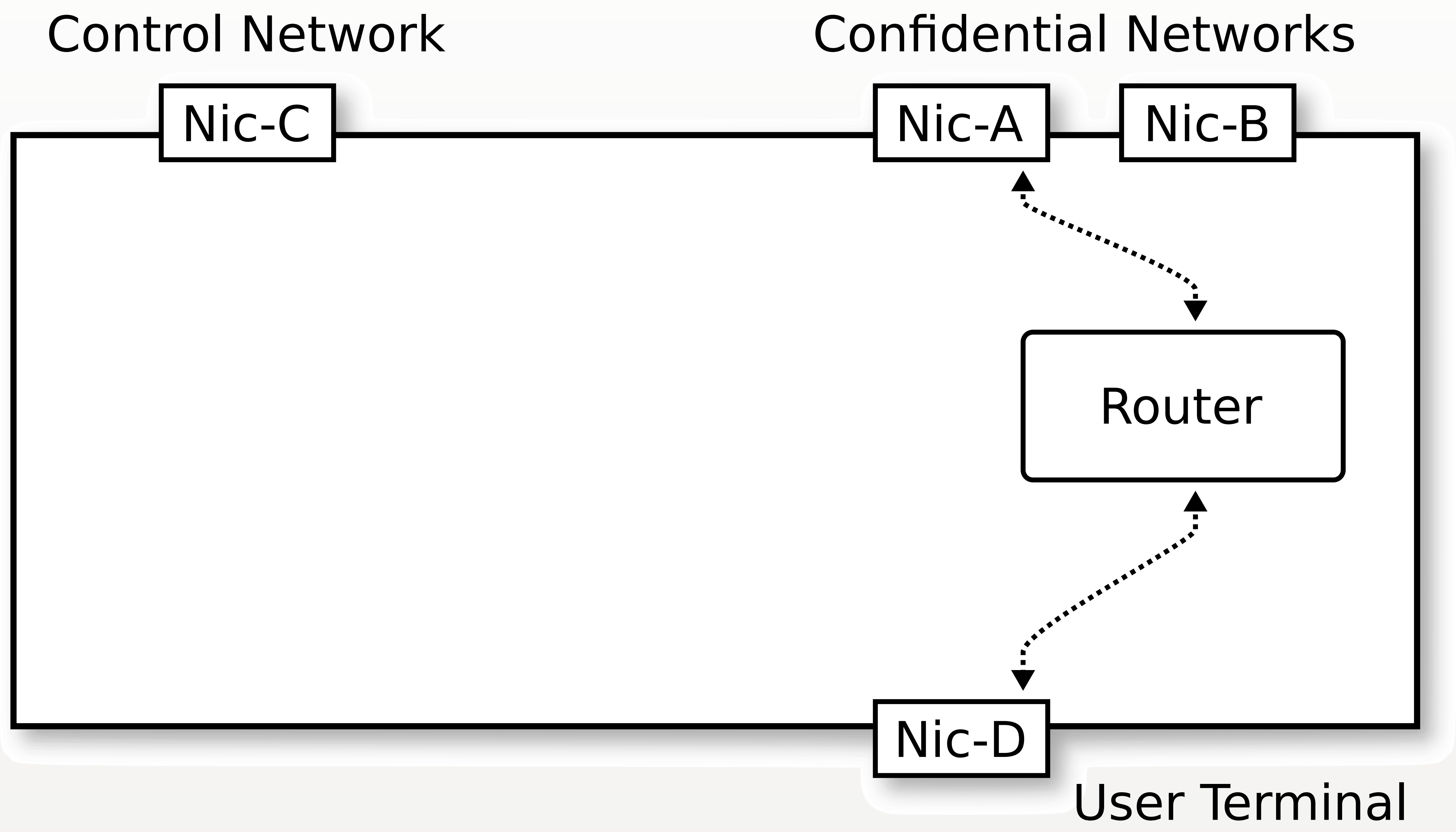
SAC Security Architecture



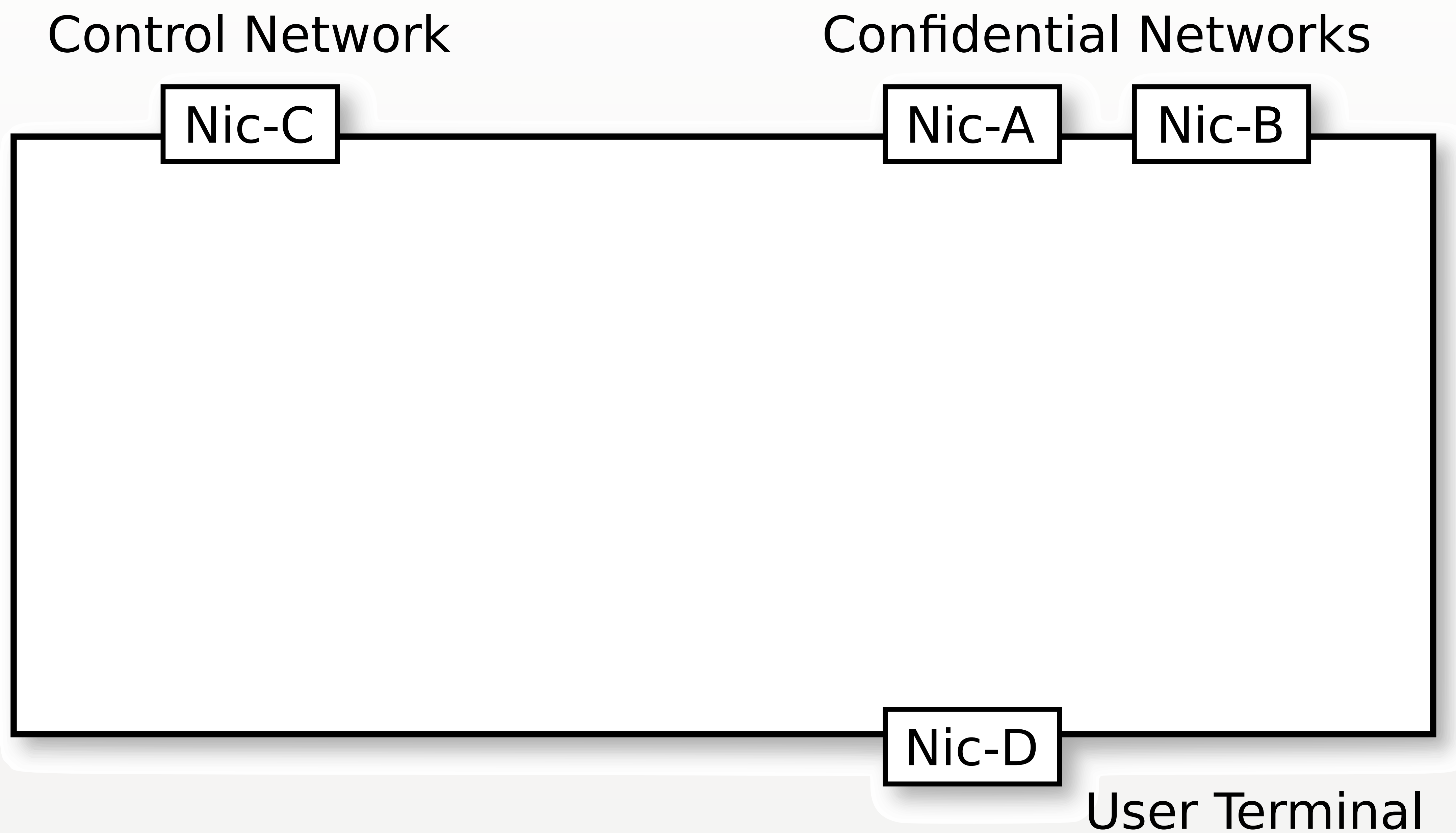
SAC Security Architecture



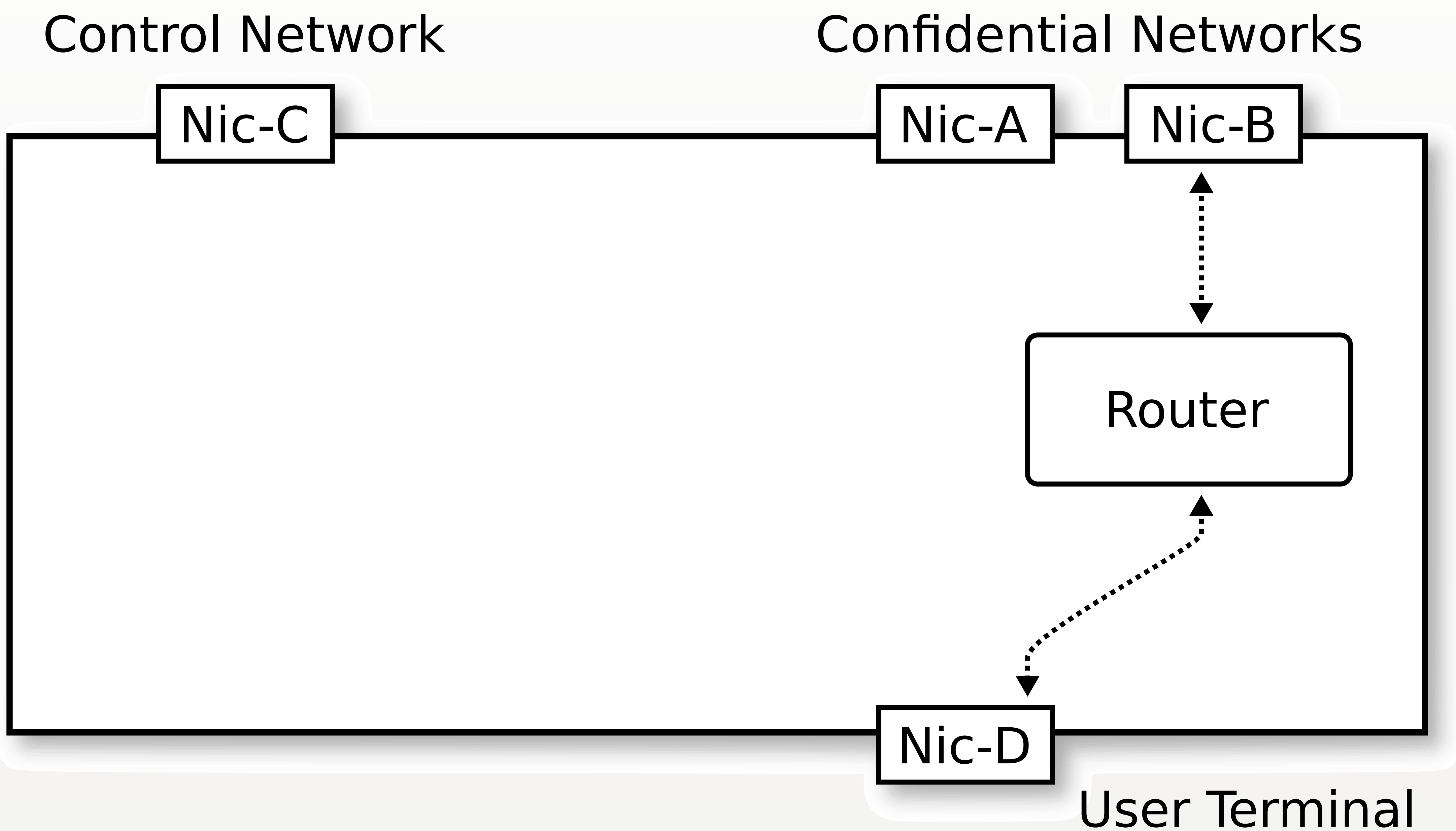
SAC Security Architecture



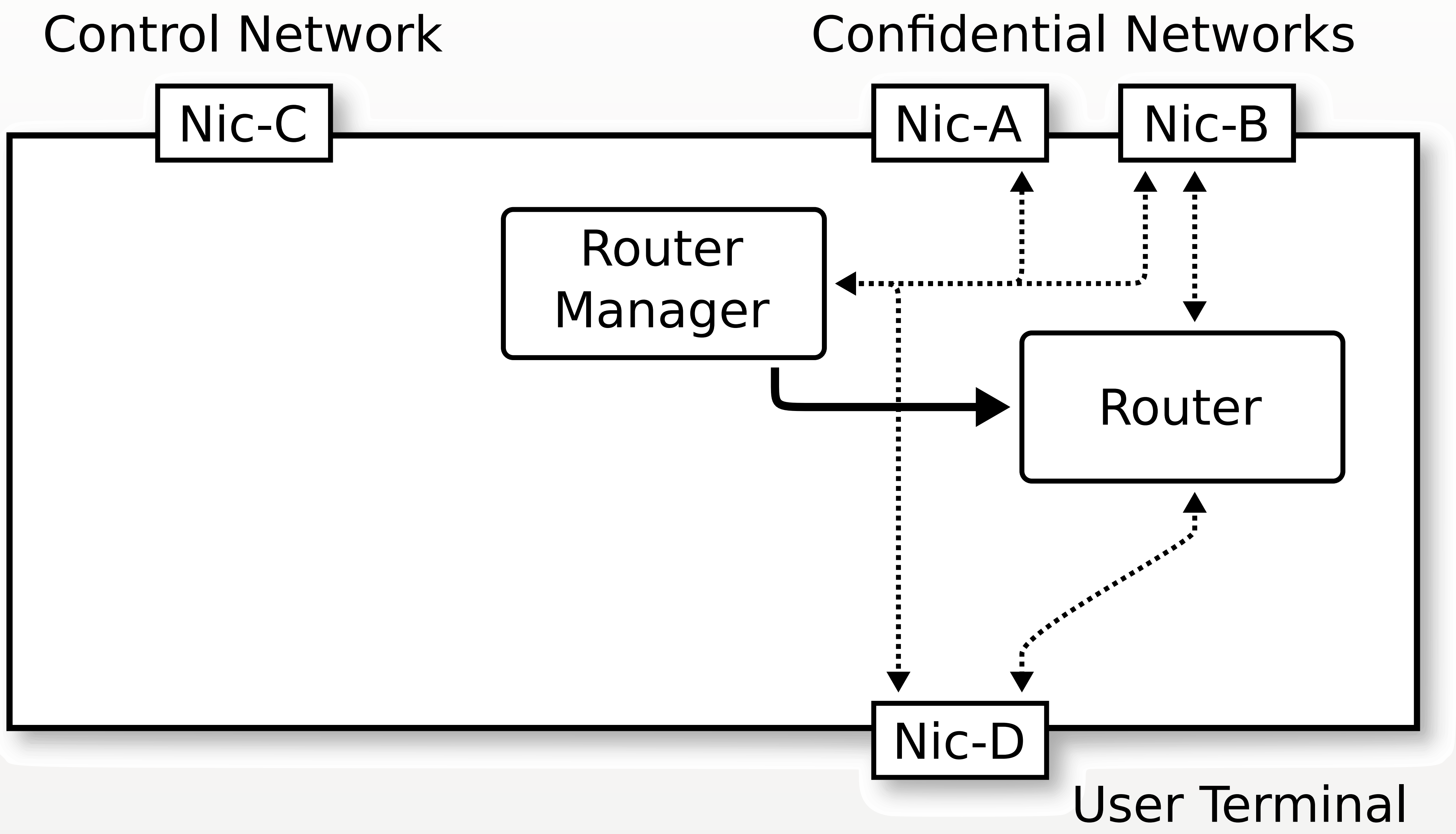
SAC Security Architecture



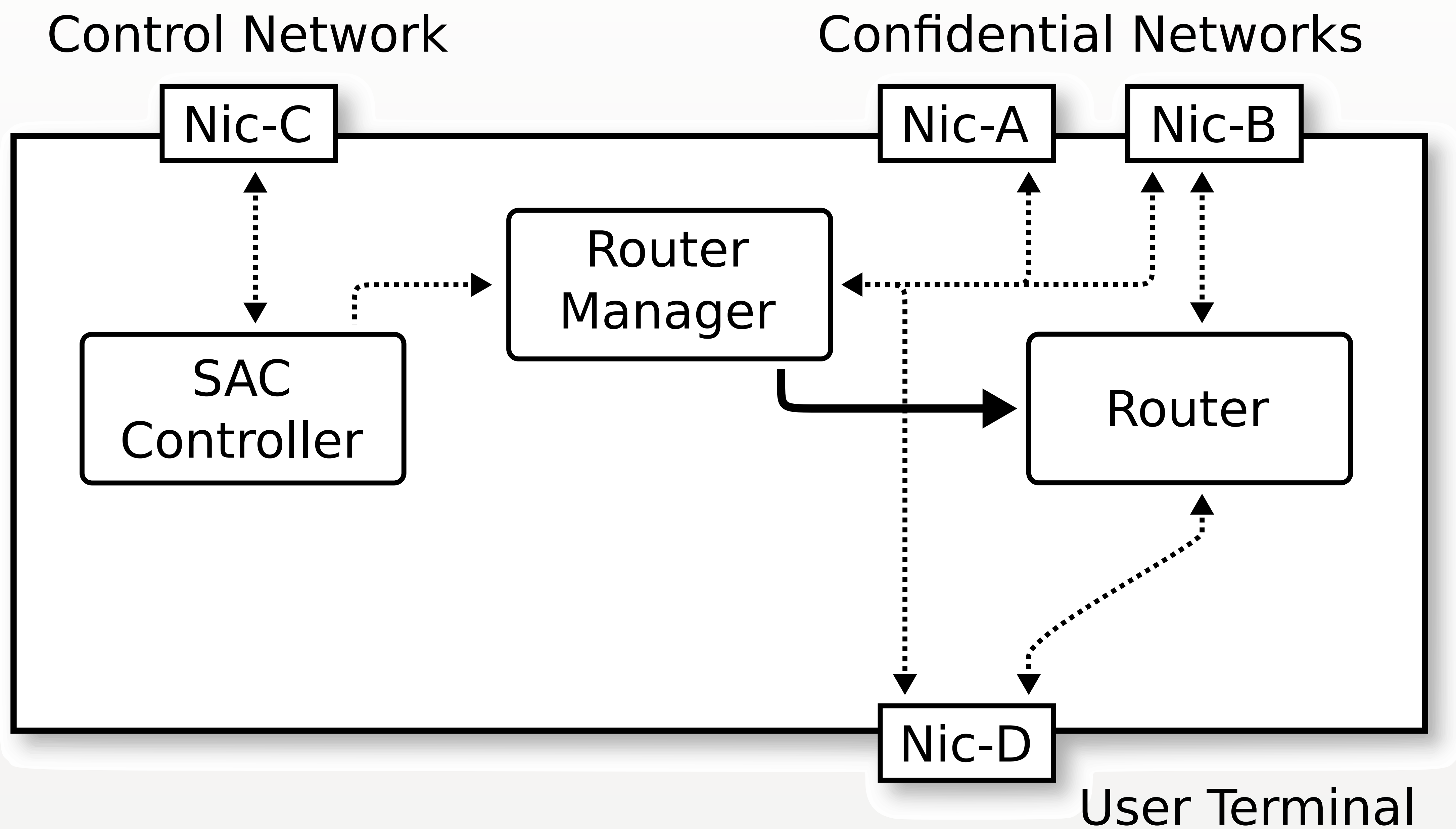
SAC Security Architecture



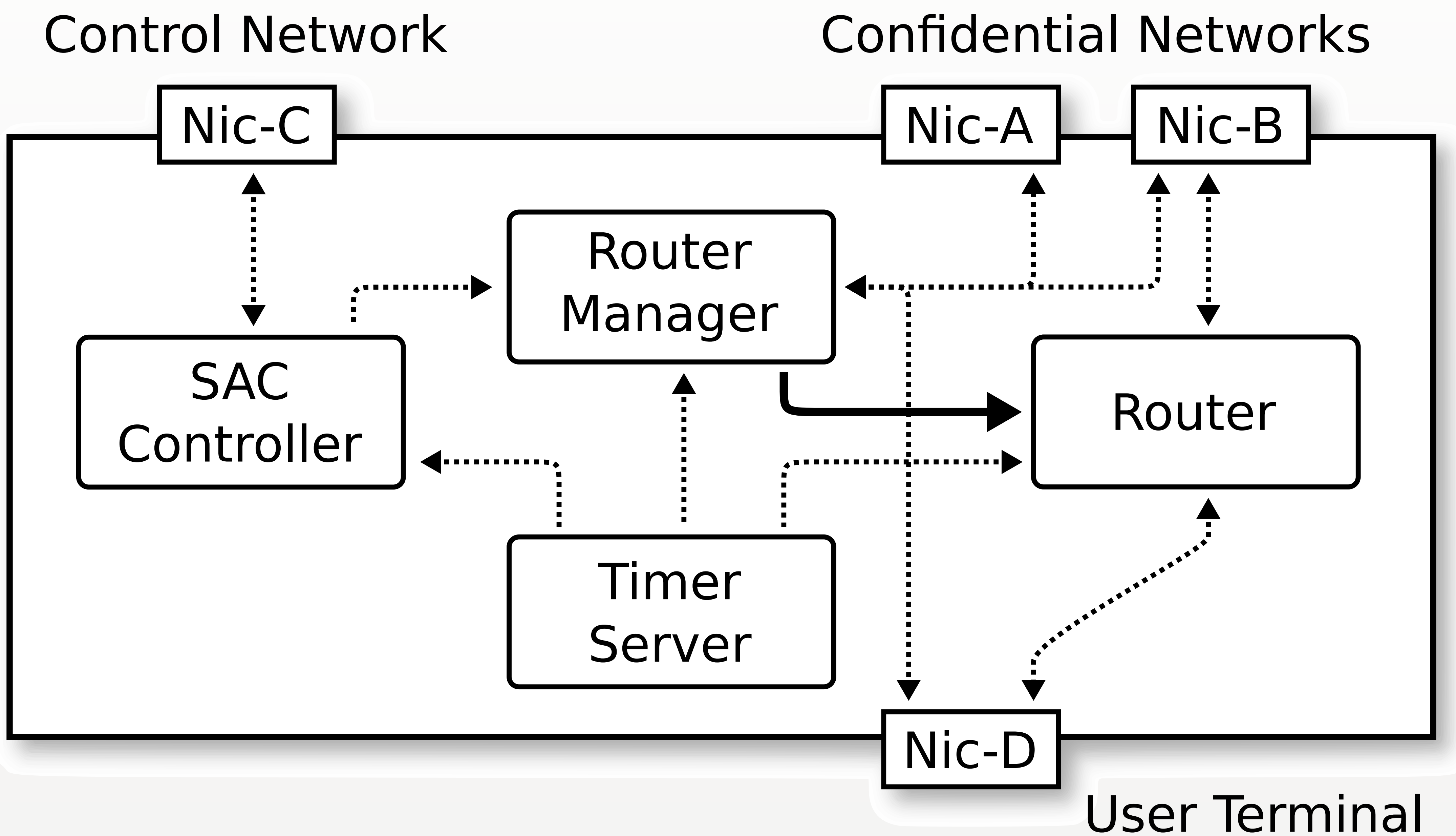
SAC Security Architecture



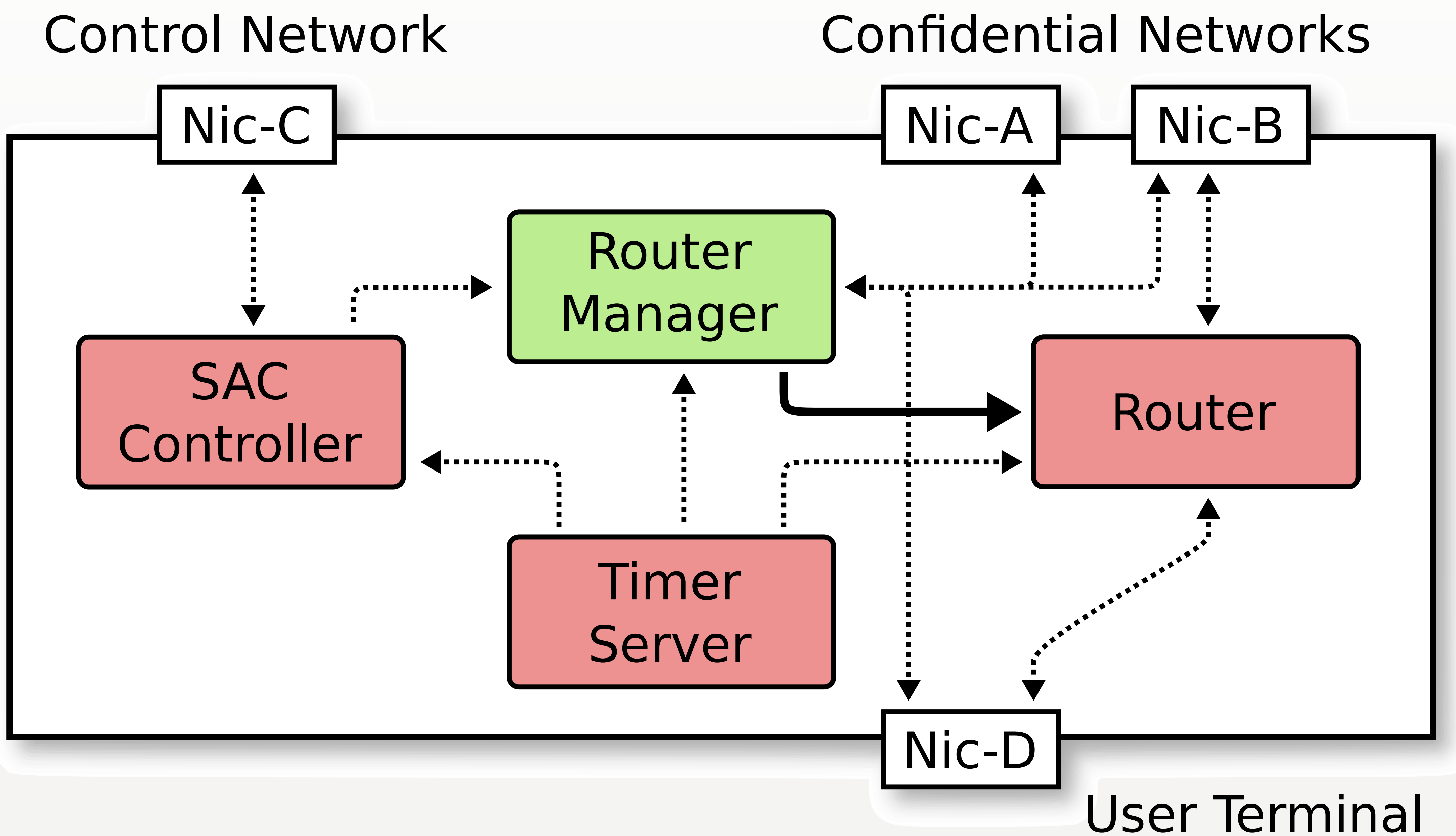
SAC Security Architecture



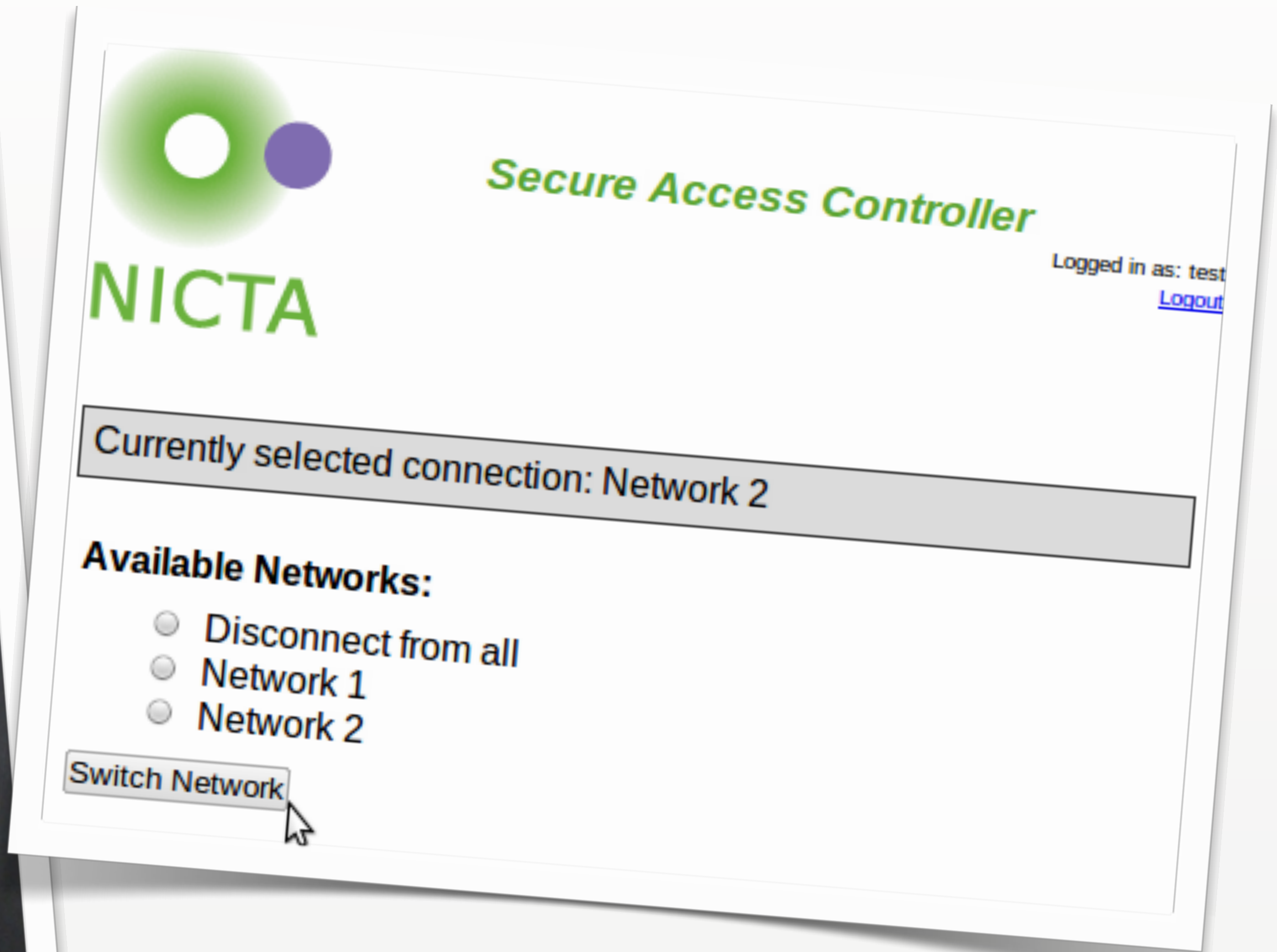
SAC Security Architecture



SAC Security Architecture

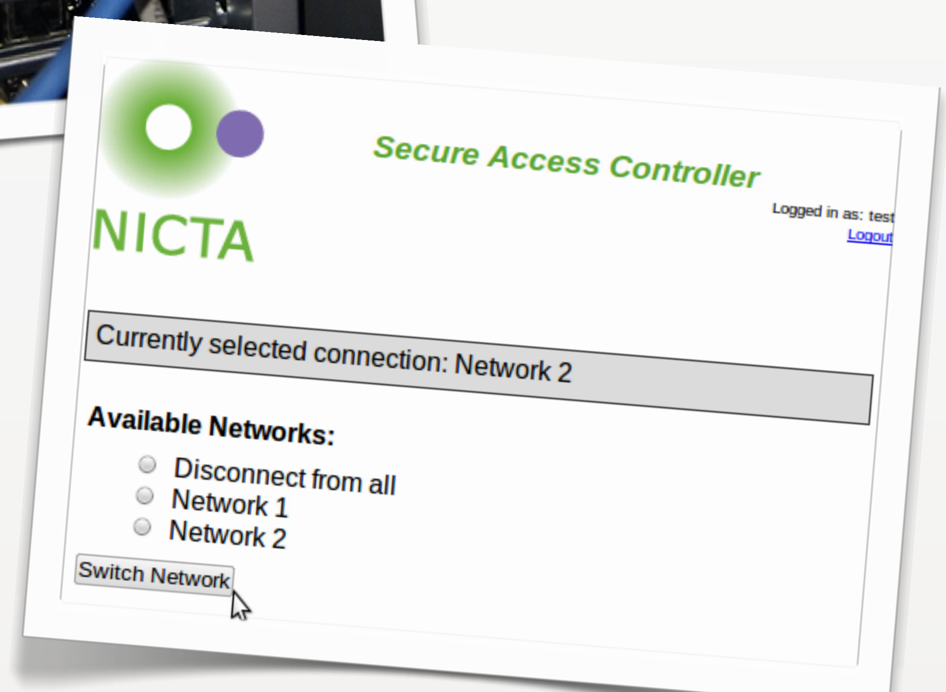


SAC Prototype



SAC Prototype

- Router
  - Virtualised Linux
  - Routing Code / NAT
- SAC Controller
  - Virtualised Linux
  - mini-httpd / OpenSSL
- Timer
  - Hand-written C
- Router Manager
  - Hand-written C
- seL4 Kernel
  - Hand-written C



SAC Prototype

- Router
  - Virtualised Linux
  - Routing Code / NAT

  } 10,000,000 LoC
- SAC Controller
  - Virtualised Linux
  - mini-httpd / OpenSSL

  } 10,000,000 LoC
- Timer
  - Hand-written C

  } 300 LoC
- Router Manager
  - Hand-written C

  } 1500 LoC
- seL4 Kernel
  - Hand-written C

  } 8300 LoC

SAC Prototype

NICTA

- Router
  - Virtualised Linux
  - Routing Code / NAT

10,000,000 LoC

- SAC Controller
  - Virtualised Linux
  - mini-httpd / OpenSSL

10,000,000 LoC

~20,000,000 lines of code

- Timer
  - Hand-written C

300 LoC

- Router Manager
  - Hand-written C

1500 LoC

- seL4 Kernel
  - Hand-written C

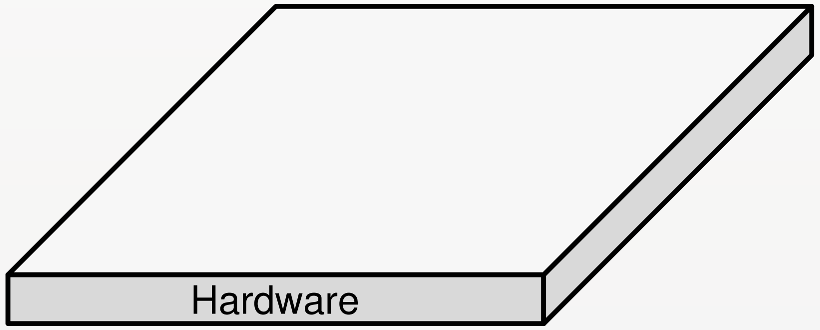
8300 LoC

~10,000 lines of code

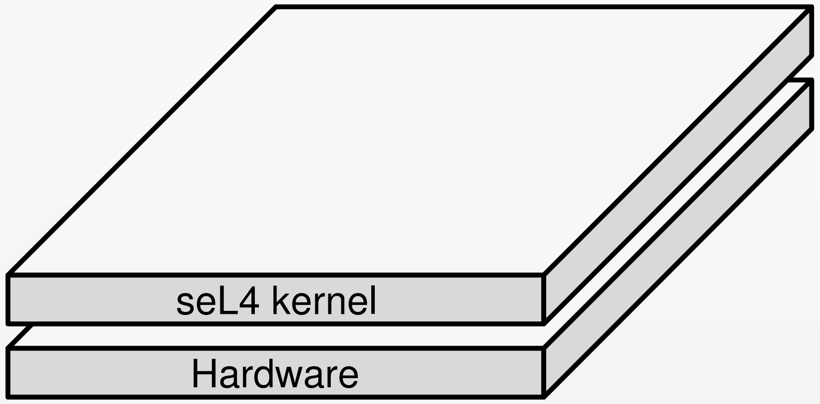
Full System Verification

- Merely *reducing* the amount of code isn’t sufficient to provide any security guarantee
- Our goal is to provide a formal guarantee
- How can we achieve this?

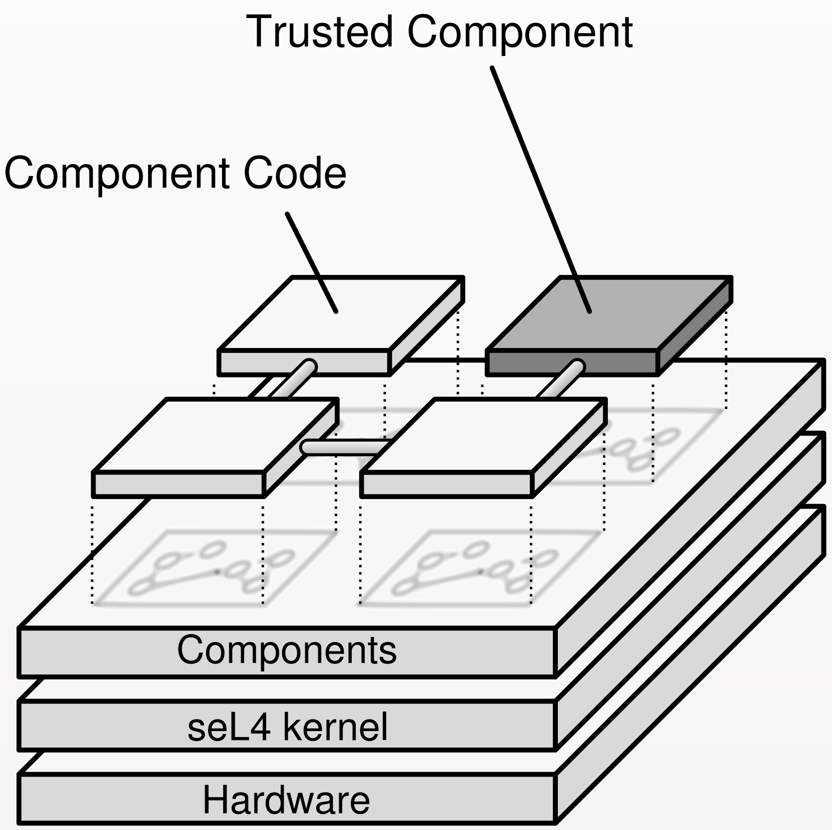
Full System Verification



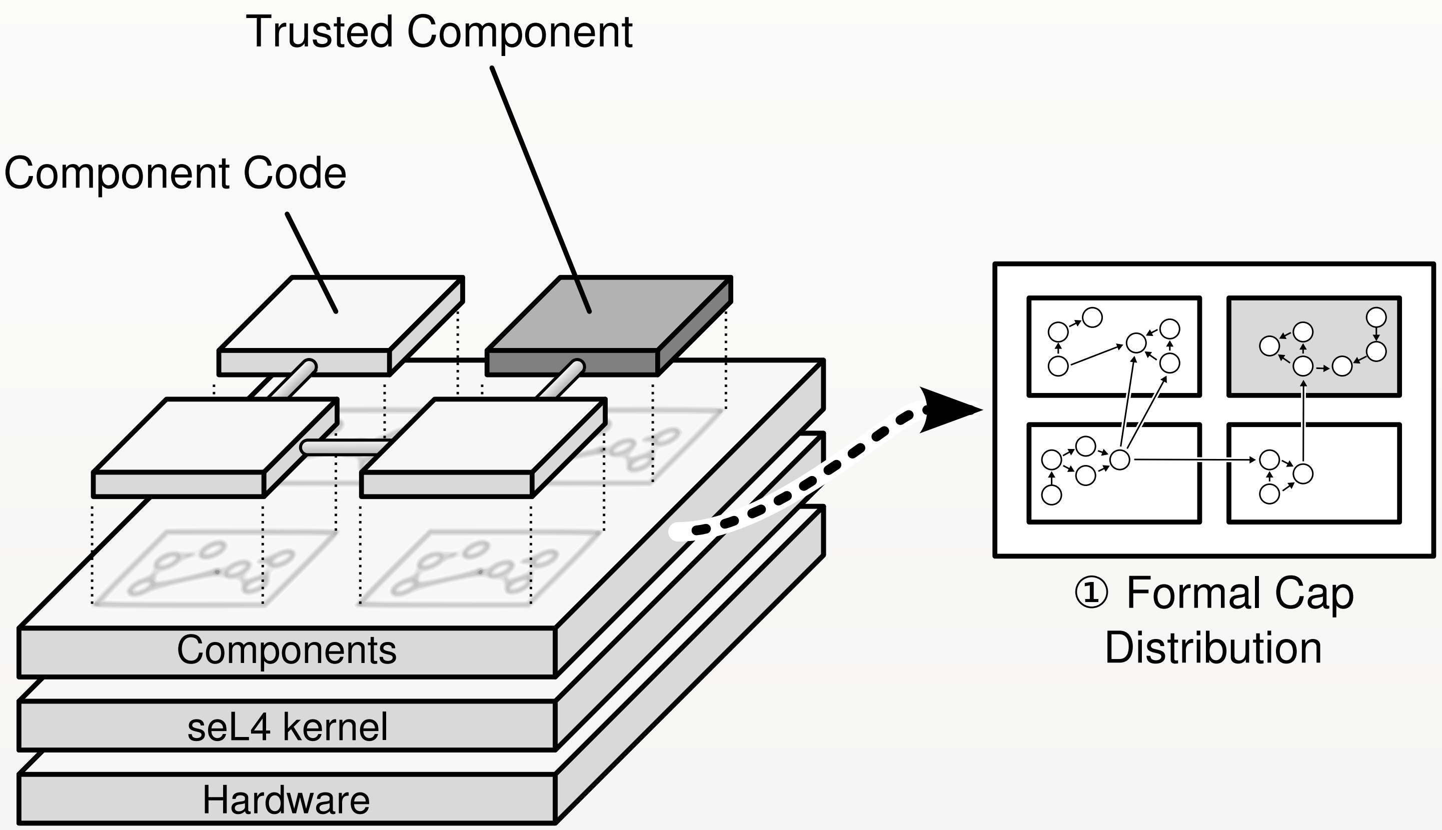
Full System Verification



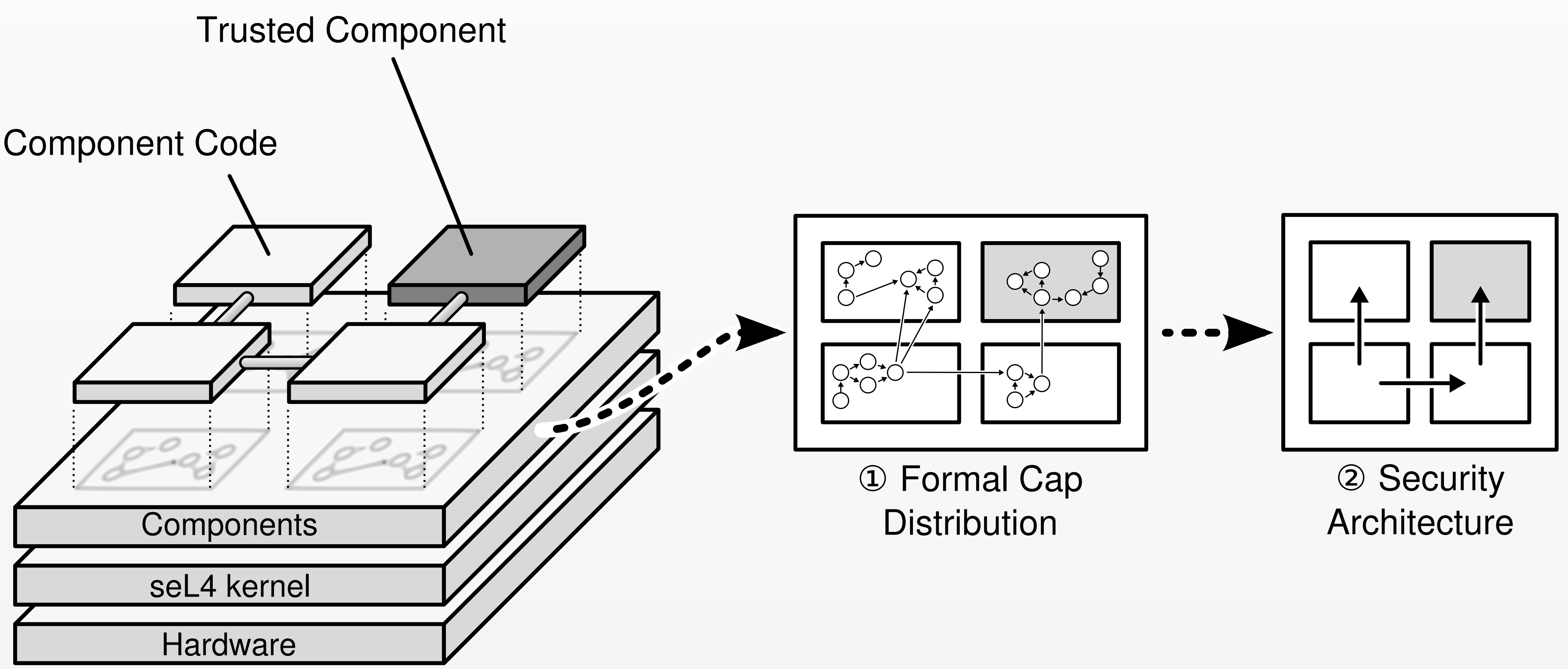
Full System Verification



Full System Verification

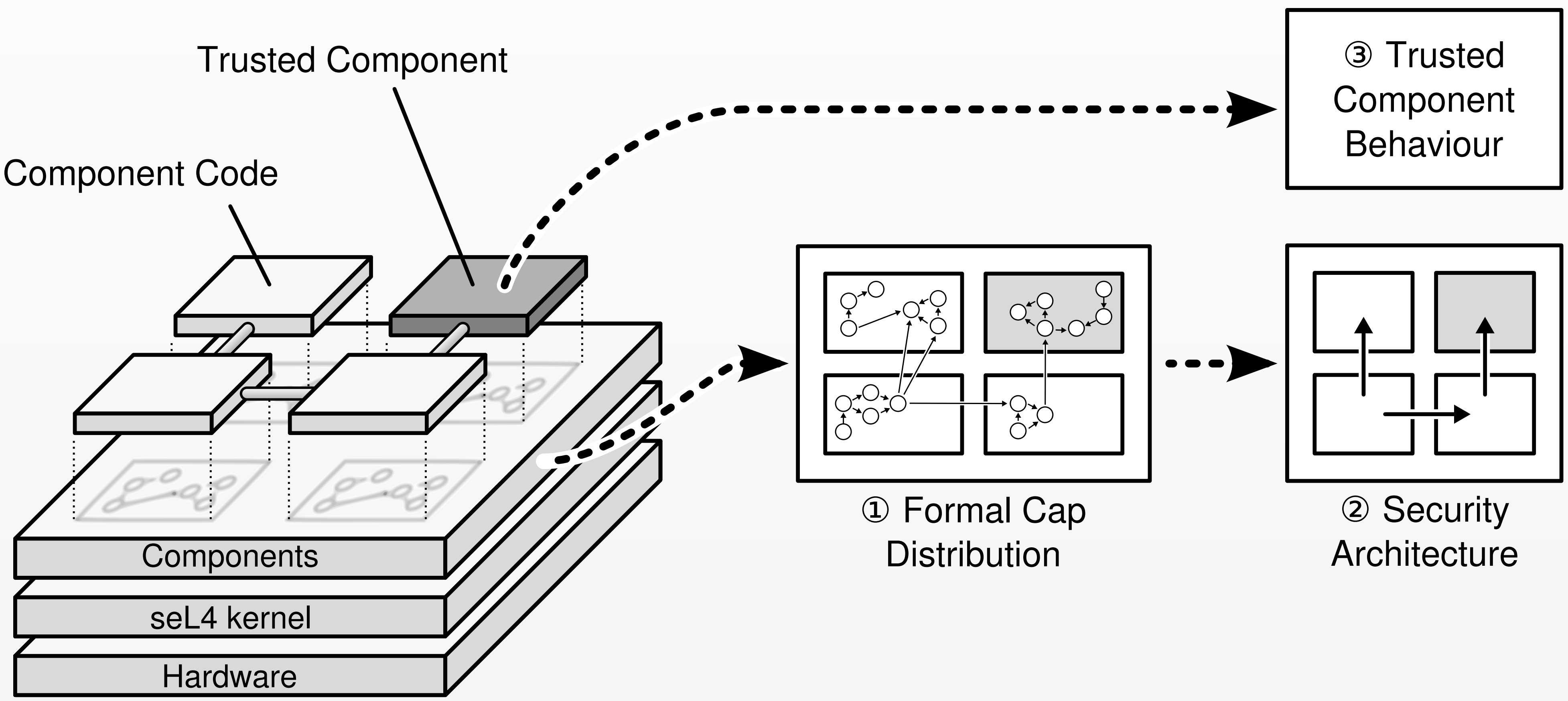


Full System Verification

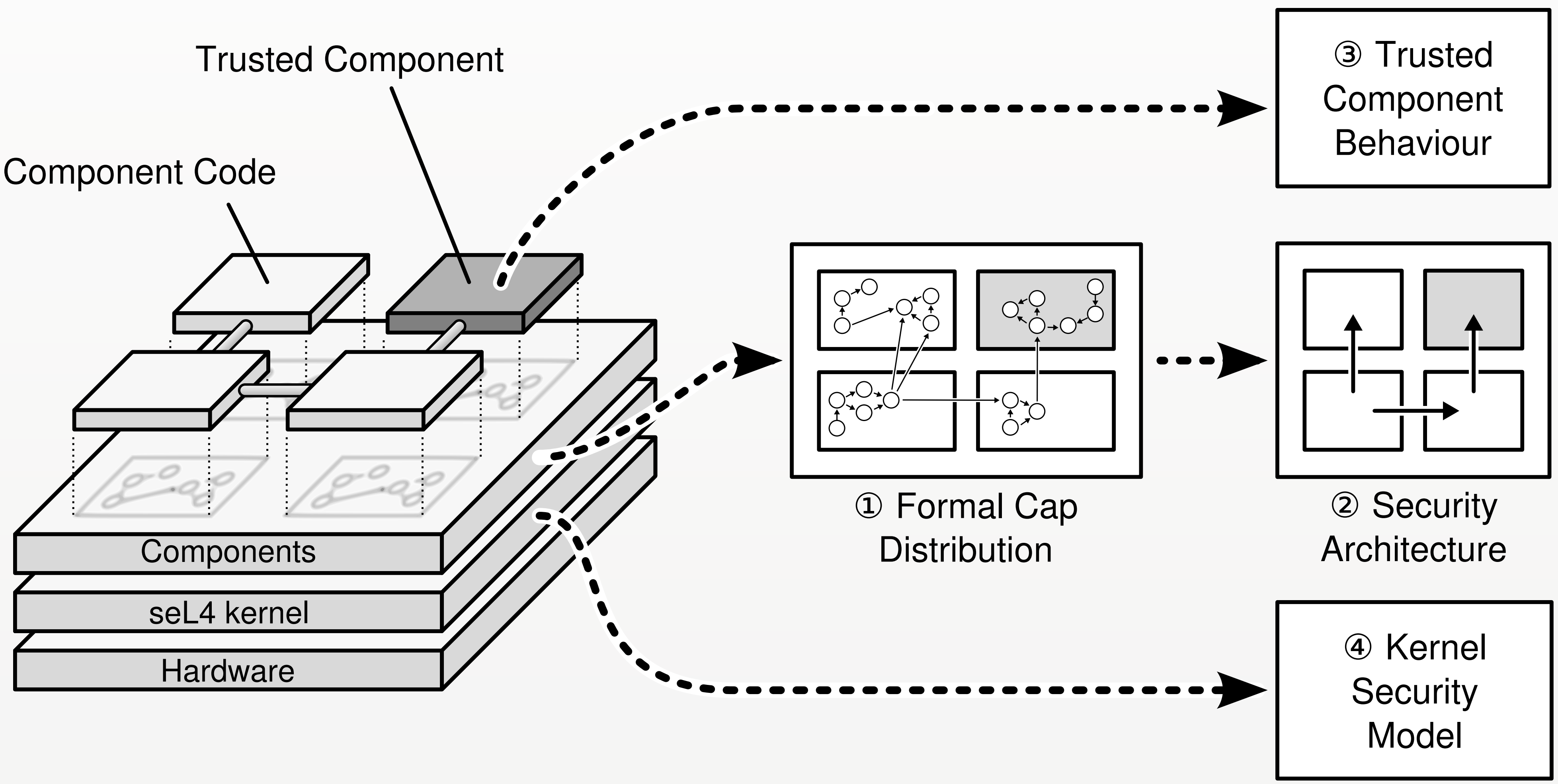


Full System Verification

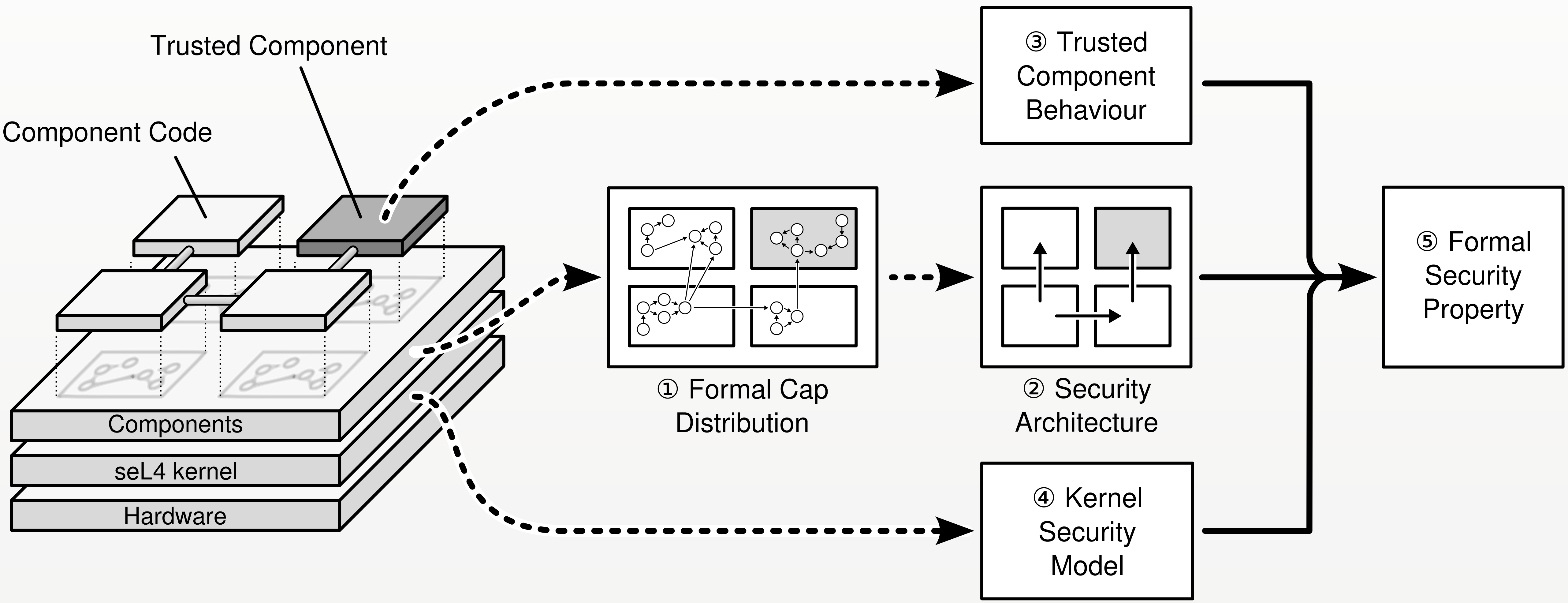
NICTA



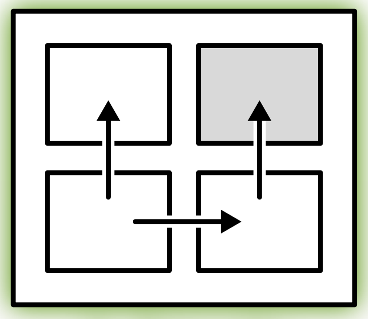
Full System Verification



Full System Verification



High Level System Model

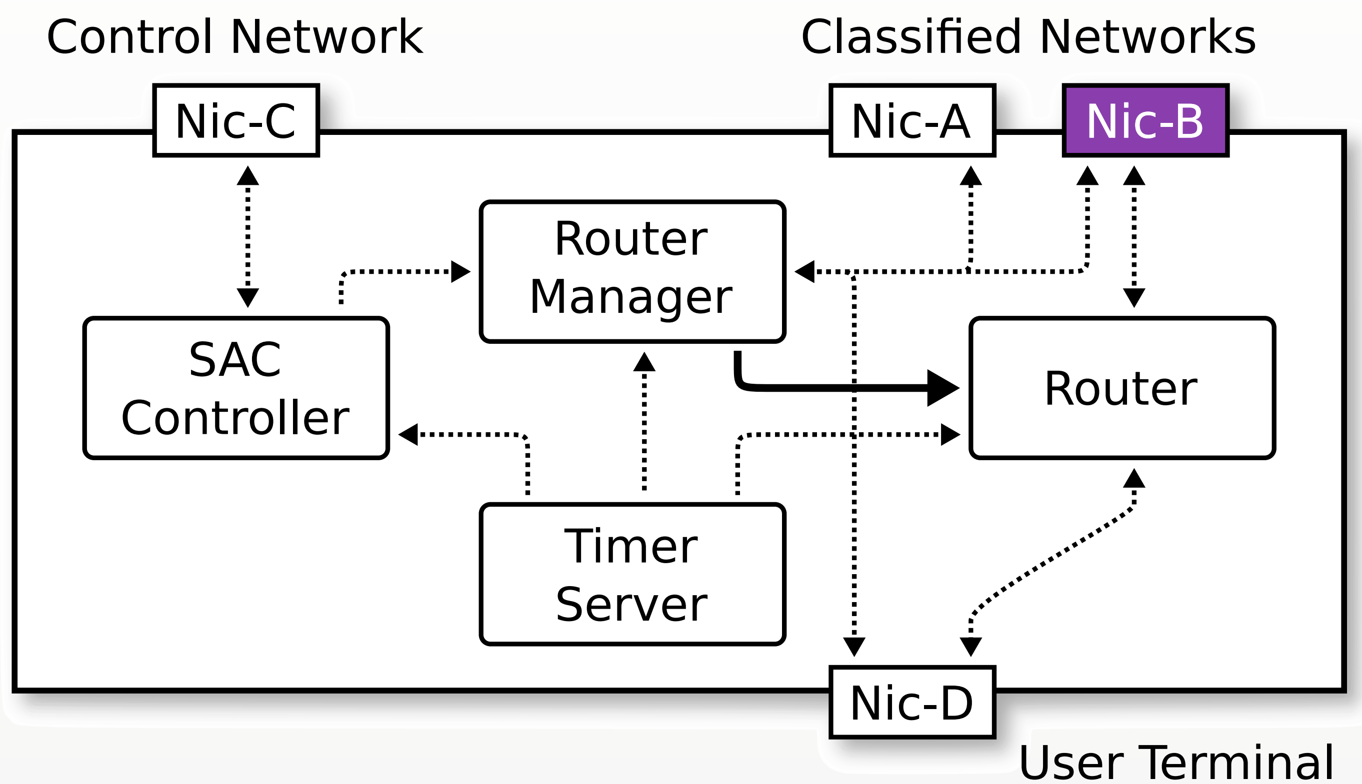


② Security Architecture

③ Trusted Component Behaviour

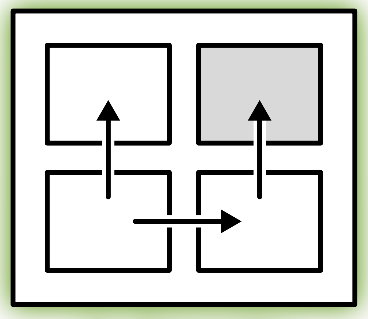
④ Kernel Security Model

⑤ Formal Security Property




```
RM_id      -> Some ({rw_to_NIC_A, rw_to_NIC_B, ...}, not_contaminated)
SAC_C_id   -> Some ({rw_to_NIC_C, w_to_RM, ...}, not_contaminated)
TIMER_id   -> Some ({w_to_SAC_C, w_to_RM, ...}, not_contaminated)
ROUTER_id  -> None
NIC_A_id   -> Some ({}, not_contaminated)
NIC_B_id   -> Some ({}, contaminated)
NIC_C_id   -> Some ({}, not_contaminated)
NIC_D_id   -> Some ({}, not_contaminated)
```

High Level System Model

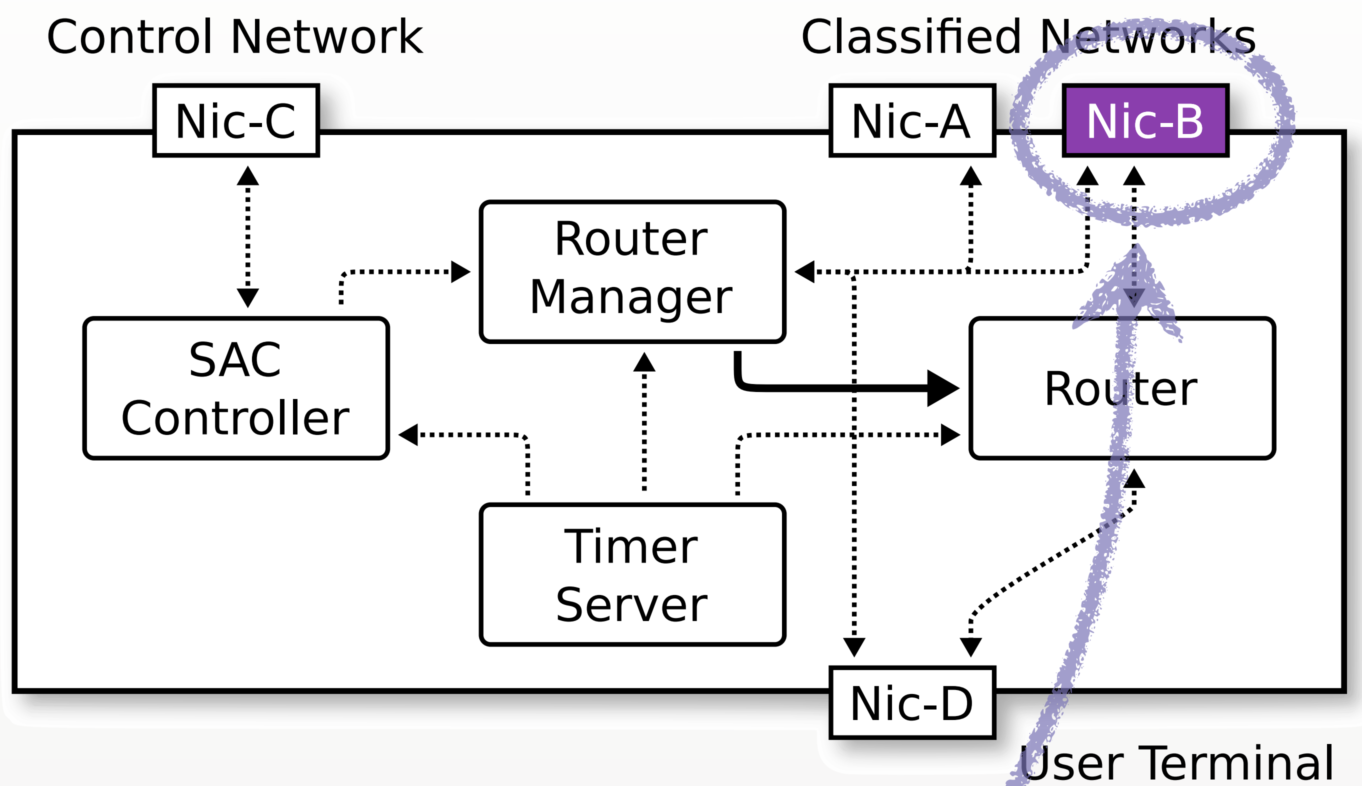


② Security Architecture

③ Trusted Component Behaviour

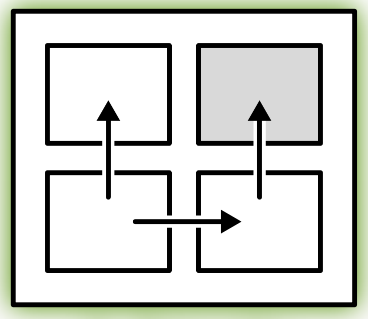
④ Kernel Security Model

⑤ Formal Security Property




```
RM_id      -> Some ({rw_to_NIC_A, rw_to_NIC_B, ...}, not_contaminated)
SAC_C_id   -> Some ({rw_to_NIC_C, w_to_RM, ...}, not_contaminated)
TIMER_id   -> Some ({w_to_SAC_C, w_to_RM, ...}, not_contaminated)
ROUTER_id  -> None
NIC_A_id   -> Some ({}, not_contaminated)
NIC_B_id   -> Some ({}, contaminated)
NIC_C_id   -> Some ({}, not_contaminated)
NIC_D_id   -> Some ({}, not_contaminated)
```

High Level System Model

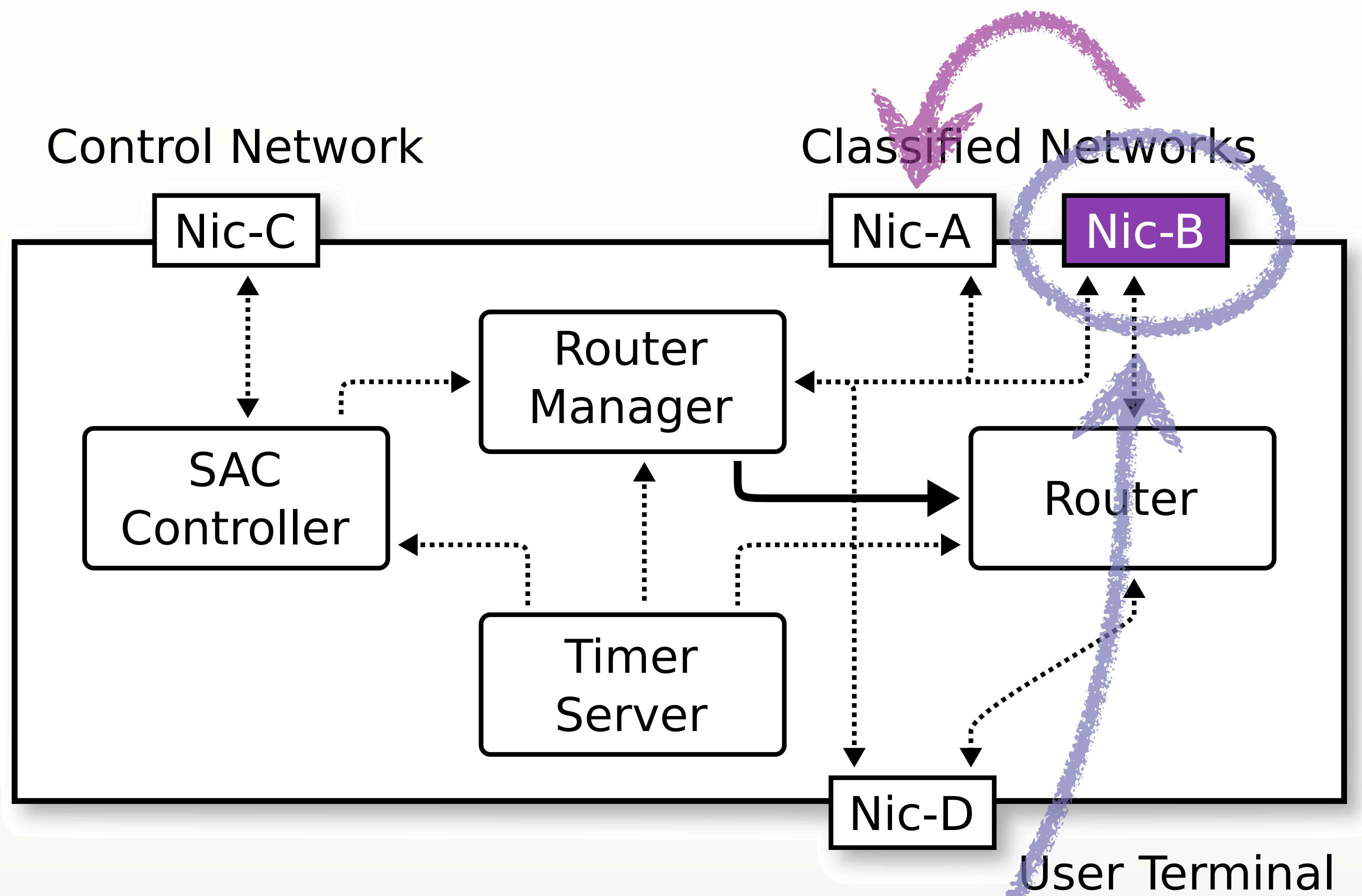


② Security Architecture

③ Trusted Component Behaviour

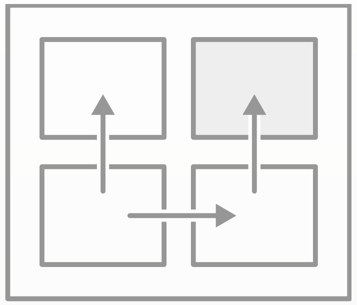
④ Kernel Security Model

⑤ Formal Security Property




```
RM_id      -> Some ({rw_to_NIC_A, rw_to_NIC_B, ...}, not_contaminated)
SAC_C_id   -> Some ({rw_to_NIC_C, w_to_RM, ...}, not_contaminated)
TIMER_id   -> Some ({w_to_SAC_C, w_to_RM, ...}, not_contaminated)
ROUTER_id  -> None
NIC_A_id   -> Some ({}, not_contaminated)
NIC_B_id   -> Some ({}, contaminated)
NIC_C_id   -> Some ({}, not_contaminated)
NIC_D_id   -> Some ({}, not_contaminated)
```

High Level System Model



② Security Architecture

③ Trusted Component Behaviour

④ Kernel Security Model

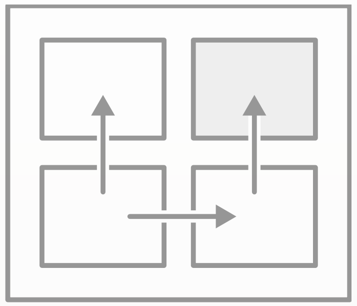
⑤ Formal Security Property

```
UNTRUSTED_prg ≡ [ AnyLegalOperation ]
```

```
RM_prg ≡
  [ (* 00: Wait for command, delete Router. *)
    SysOp (SysRead cap_R_to_SAC_C),
    SysOp (SysRemoveAll cap_C_to_R),
    SysOp (SysDelete cap_C_to_R),
    SysOp (SysWriteZero cap_RW_to_NIC_D).
    ...
    (* 09: Non-deterministic “goto” *)
    Jump [0, 10, 19],

    (* 10: Setup Router between NIC-A and NIC-D *)
    SysOp (SysCreate cap_C_to_R),
    SysOp (SysNormalWrite cap_RWGC_to_R),
    ...
  ]
```

High Level System Model



② Security Architecture

③ Trusted Component Behaviour

④ Kernel Security Model

⑤ Formal Security Property

```
step state e (SysRead c) =
  write_operation (entity c) e state
```

What operations do user system calls perform?

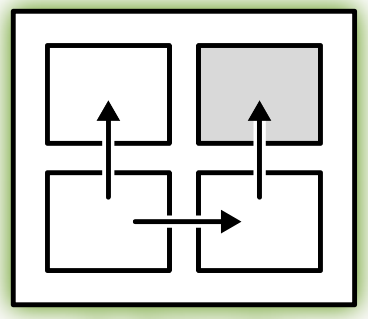
```
legal s e (SysRead cap) =
  (is_entity s e
     ∧ is_entity s (entity cap)
     ∧ cap ∈ entity_caps_in_state s e
     ∧ Read ∈ rights cap)
```

When is a system call allowed by the kernel?

```
write_operation source target ss ≡
  (case (ss target) of
    Some target_entity ⇒
      ss(target ↦ target_entity(
        contaminated :=
            is_contaminated ss target
          ∨ is_contaminated ss source)
  | _ ⇒ ss)
```

What effect do system calls have?

High Level System Model

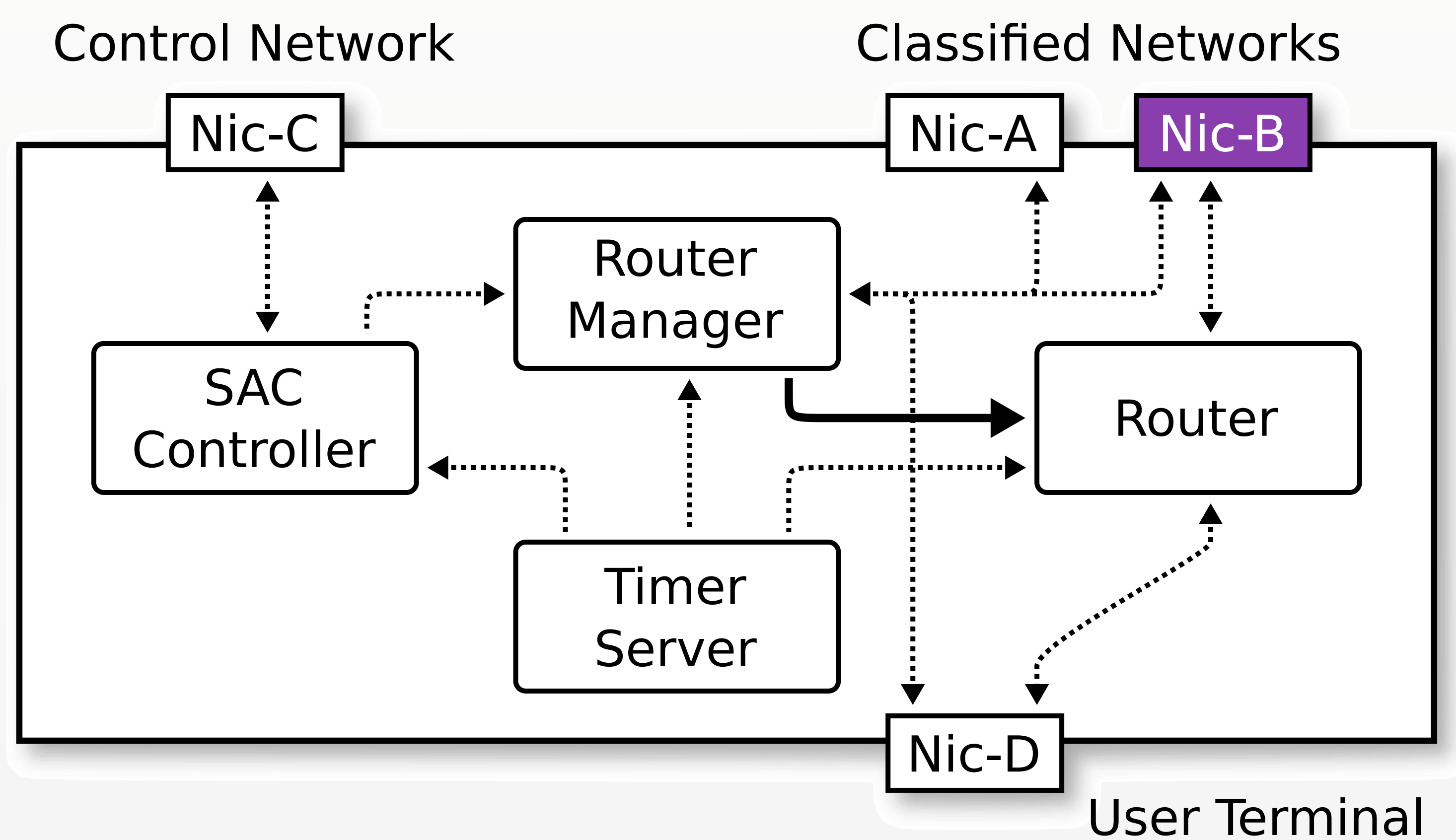


② Security Architecture

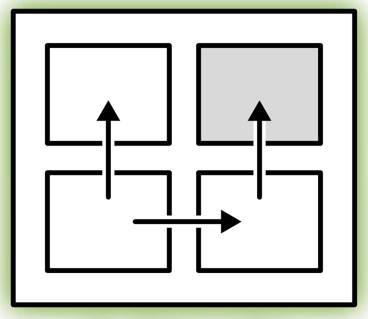
③ Trusted Component Behaviour

④ Kernel Security Model

⑤ Formal Security Property



High Level System Model

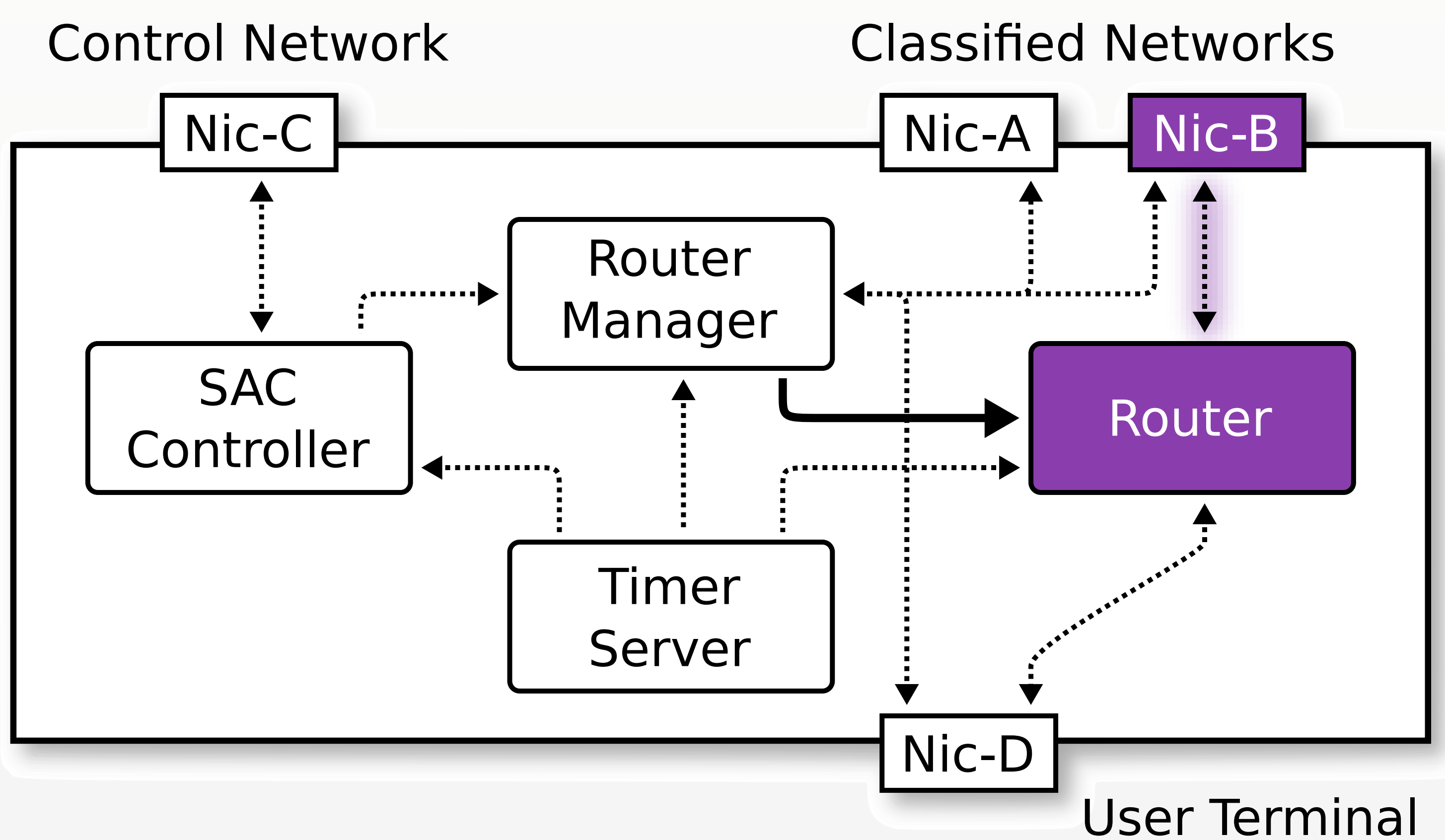


② Security Architecture

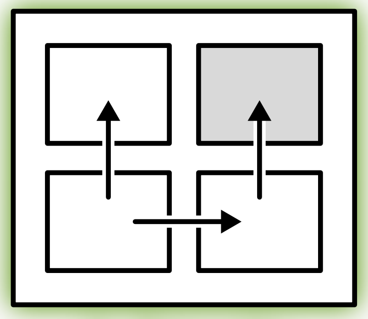
③ Trusted Component Behaviour

④ Kernel Security Model

⑤ Formal Security Property



High Level System Model

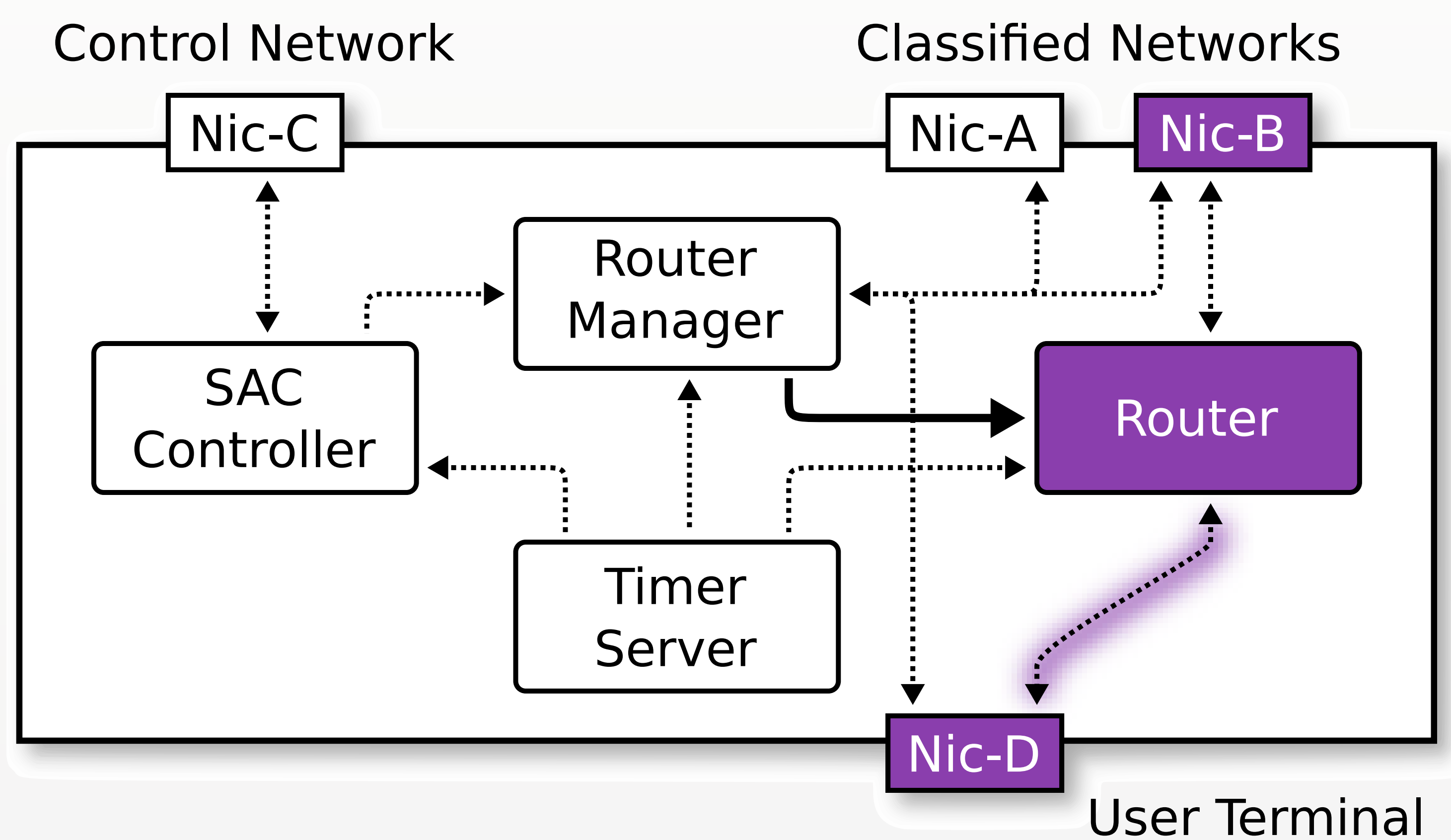


② Security Architecture

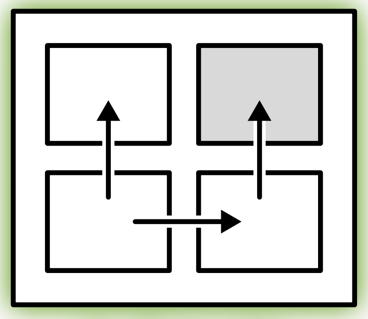
③ Trusted Component Behaviour

④ Kernel Security Model

⑤ Formal Security Property



High Level System Model

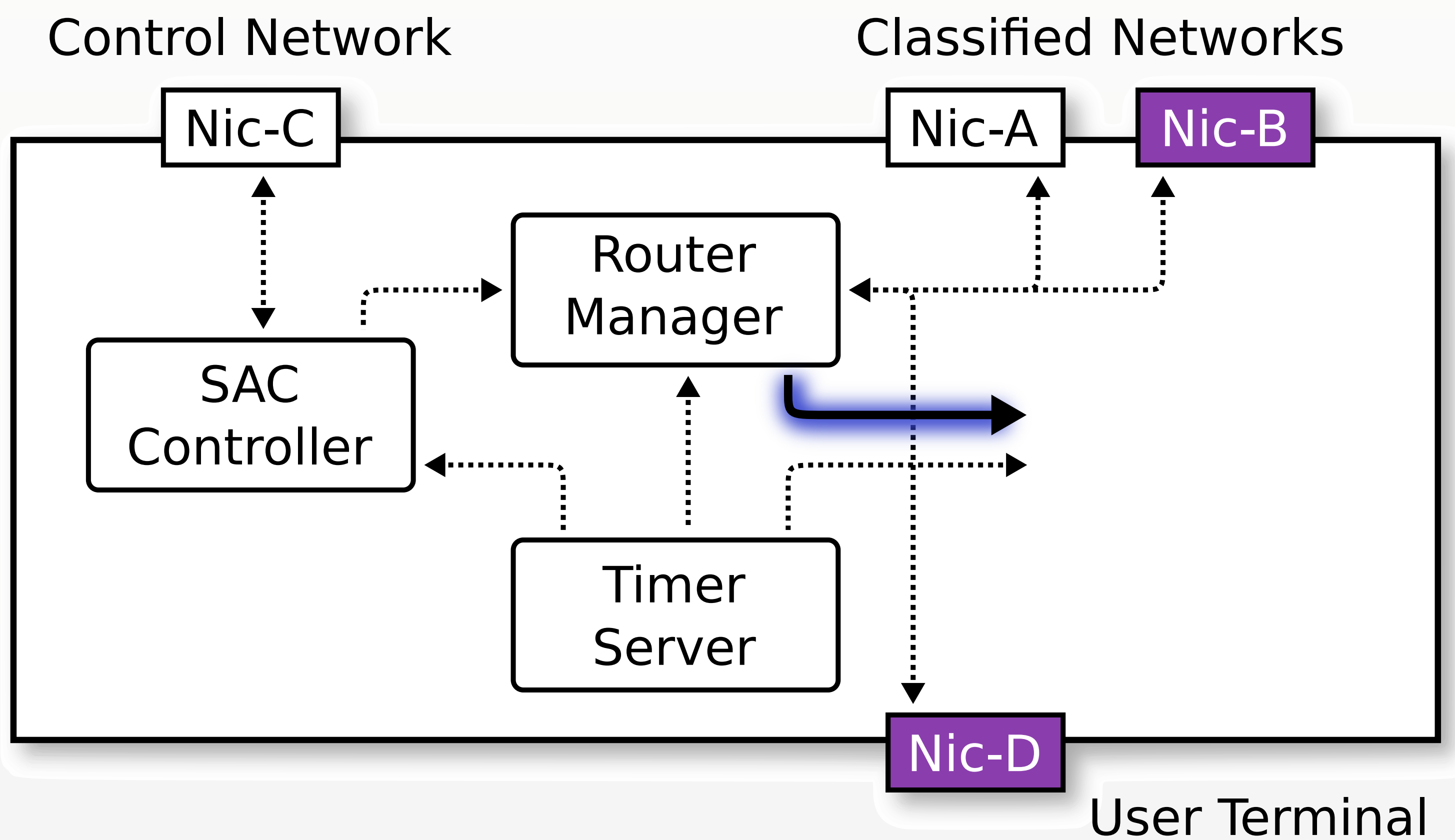


② Security Architecture

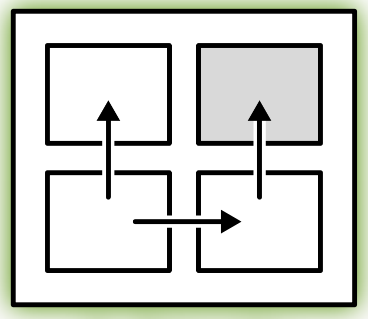
③ Trusted Component Behaviour

④ Kernel Security Model

⑤ Formal Security Property



High Level System Model

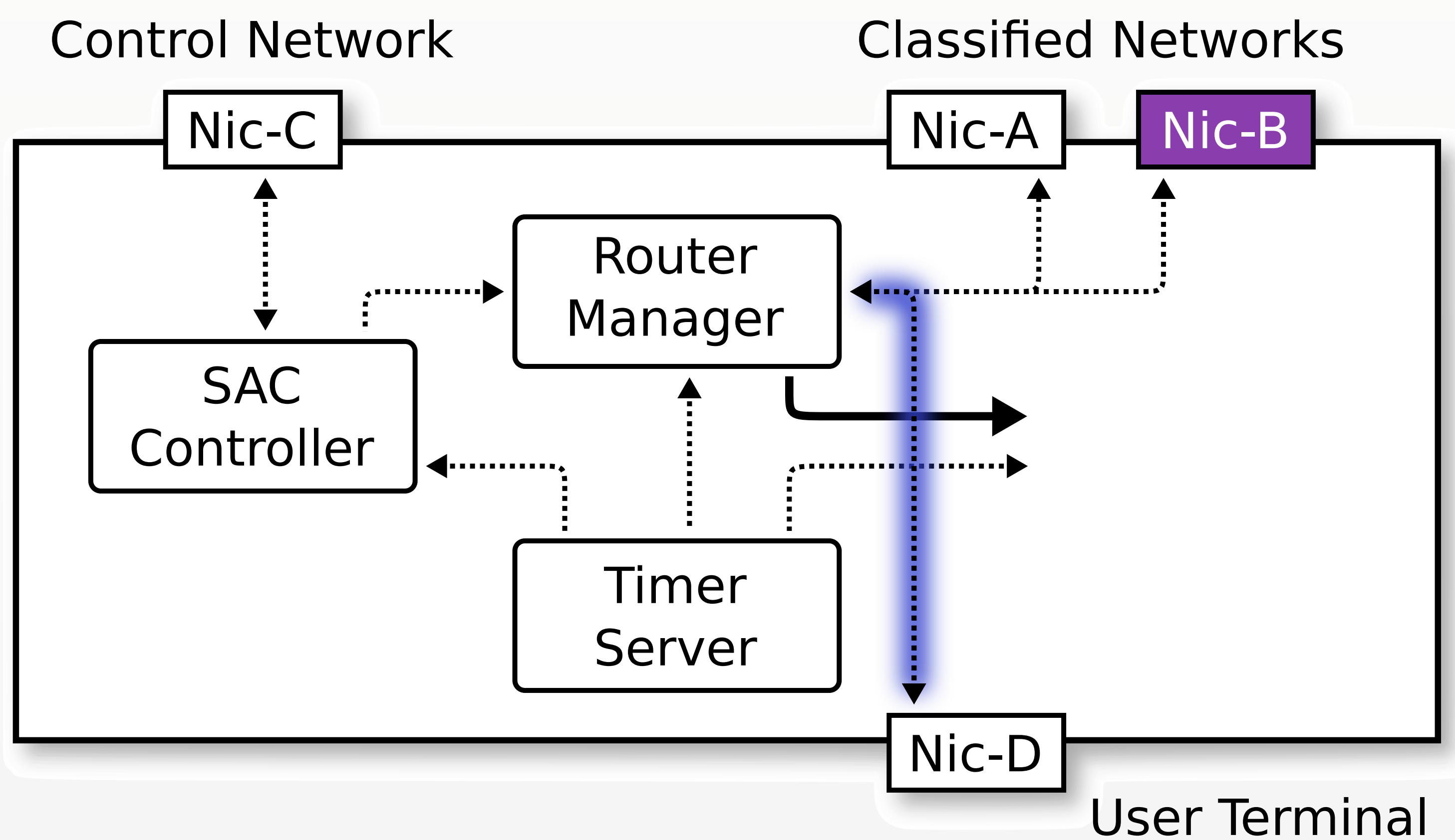


② Security Architecture

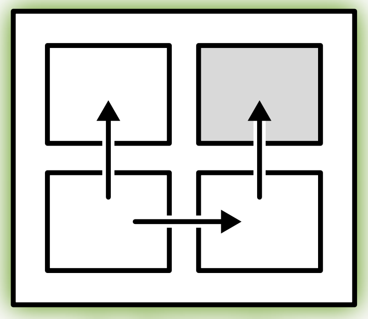
③ Trusted Component Behaviour

④ Kernel Security Model

⑤ Formal Security Property



High Level System Model

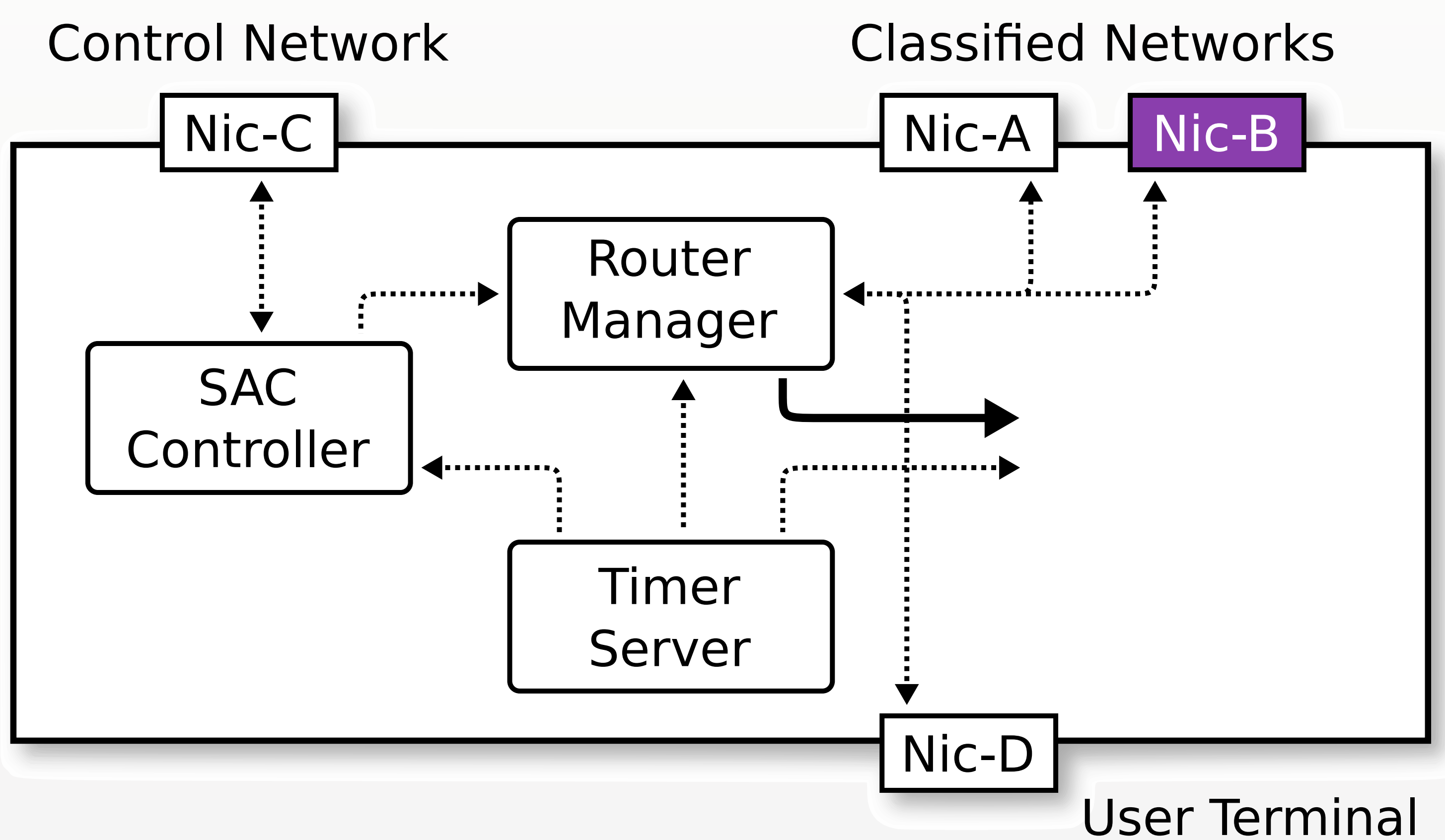


② Security Architecture

③ Trusted Component Behaviour

④ Kernel Security Model

⑤ Formal Security Property



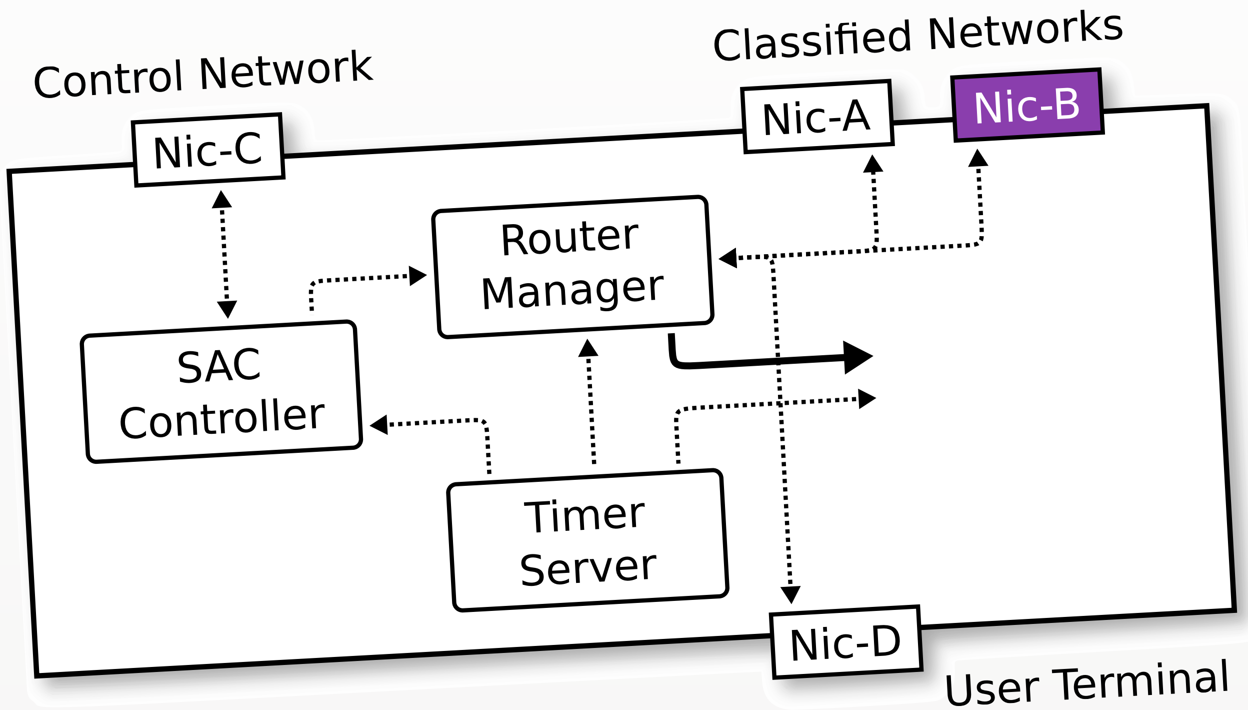
High Level System Model

② Security Architecture

③ Trusted Component Behaviour

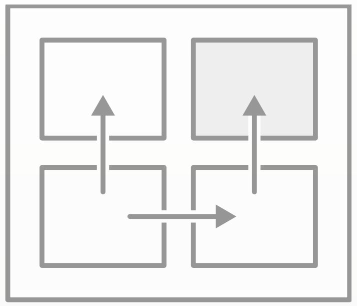
④ Kernel Security Model

⑤ Formal Security Property




```
theorem sac_is_secure:
  (SAC-startup →* ss) ⟹ ¬ is_contaminated (sac-entity-ss) NicA
```

High Level System Model

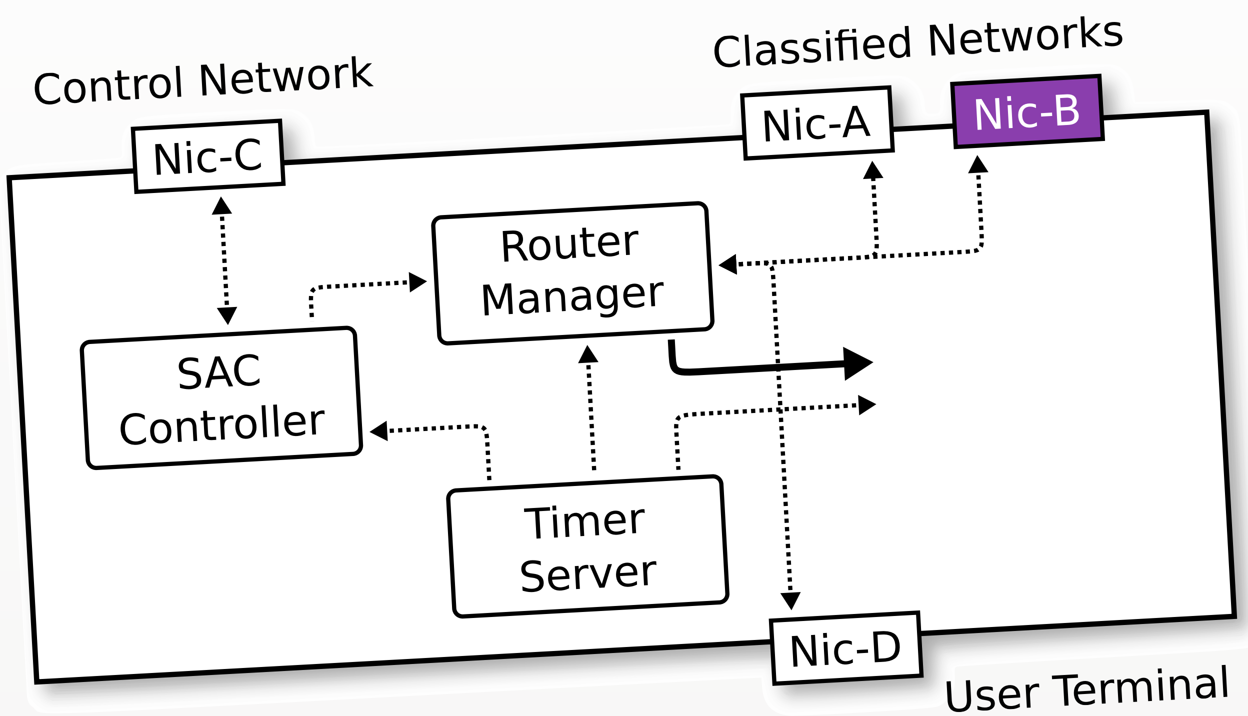


② Security Architecture

③ Trusted Component Behaviour

④ Kernel Security Model

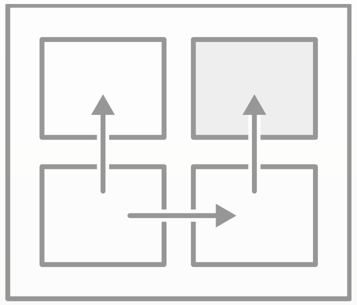
⑤ Formal Security Property




```
theorem sac_is_secure:
  (SAC-startup →* ss) ⟹ ¬ is_contaminated (sac-entity-ss) NicA
```

High Level System Model

NICTA

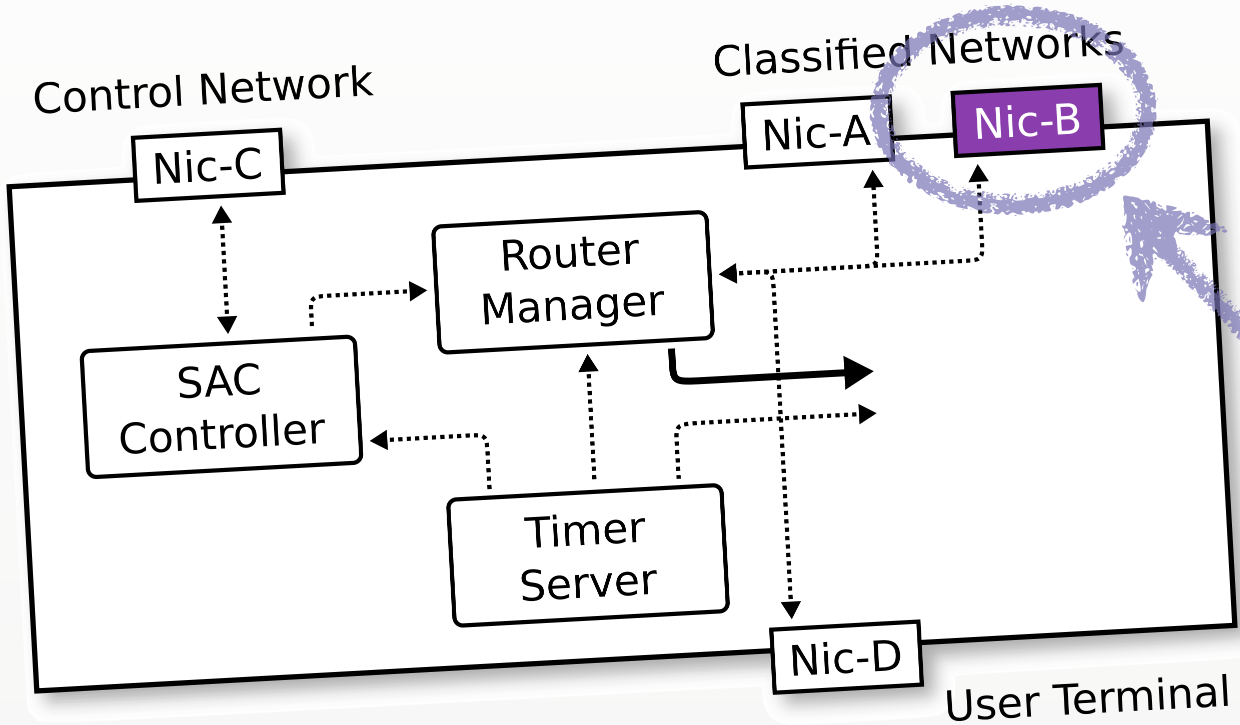


② Security Architecture

③ Trusted Component Behaviour

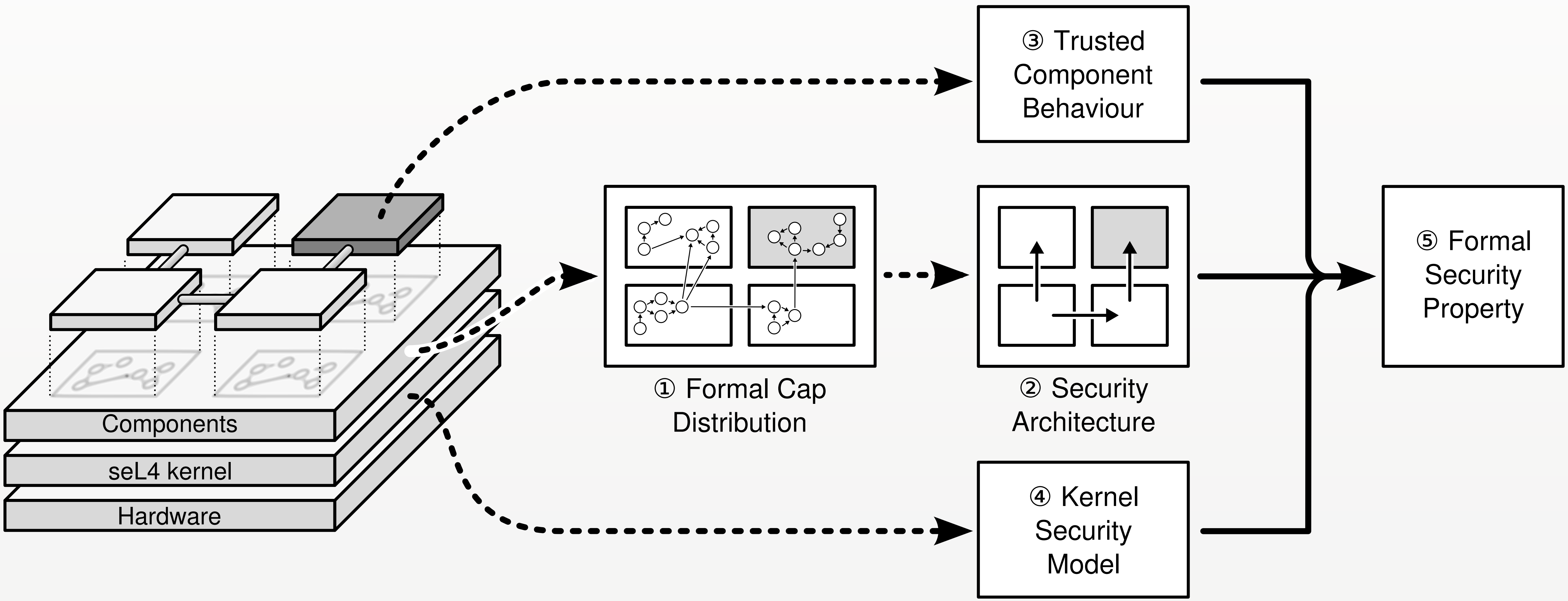
④ Kernel Security Model

⑤ Formal Security Property



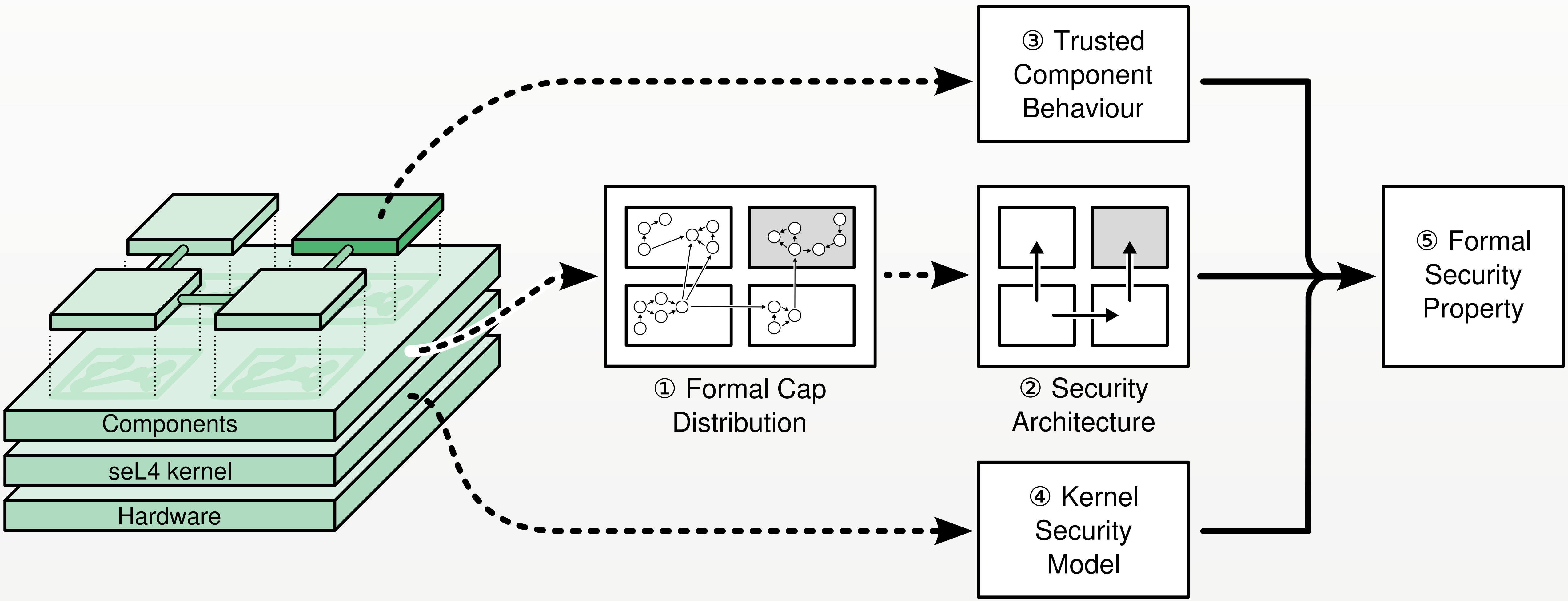

```
theorem sac_is_secure:
  (SAC-startup →* ss) ⟹ ¬ is_contaminated (sac-entity-ss) NicA
```

Progress

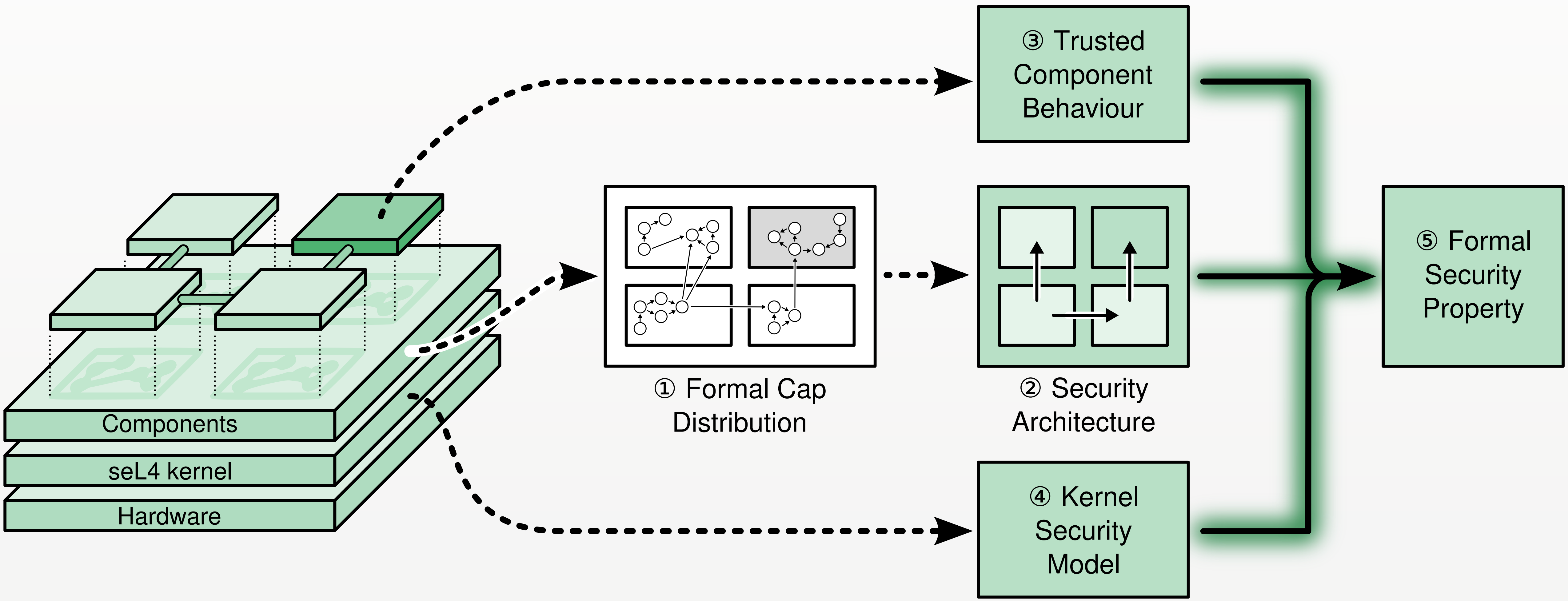


③ Trusted Component Behaviour
Components
seL4 kernel
Hardware
① Formal Cap Distribution
② Security Architecture
⑤ Formal Security Property
④ Kernel Security Model

Progress

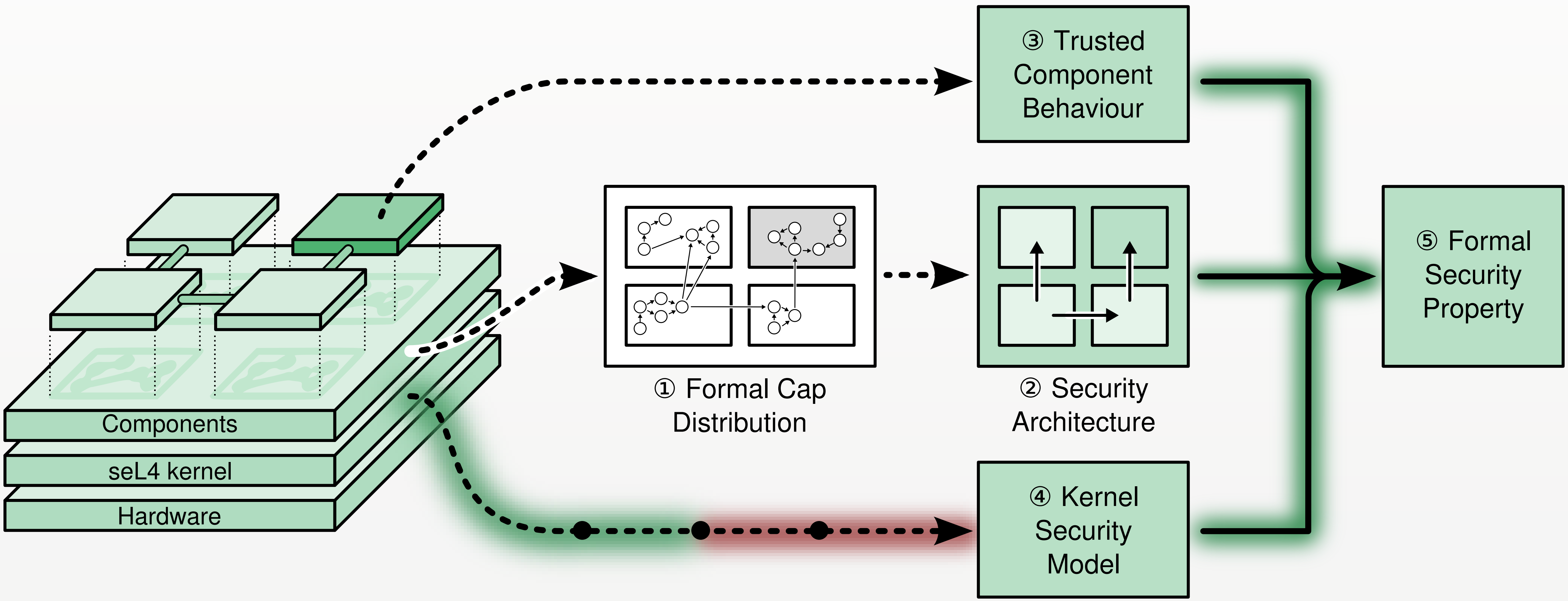

Components
seL4 kernel
Hardware
① Formal Cap
Distribution
② Security
Architecture
③ Trusted
Component
Behaviour
④ Kernel
Security
Model
⑤ Formal
Security
Property

Progress



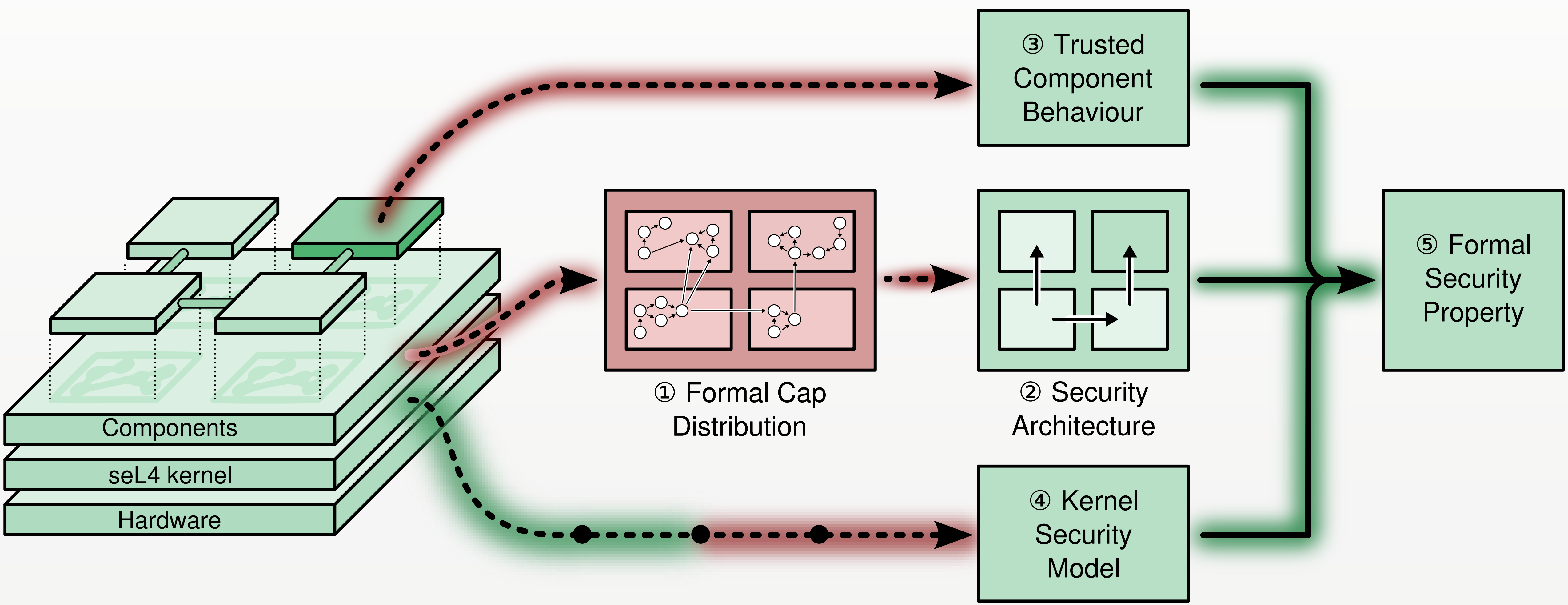
③ Trusted Component Behaviour
Components
seL4 kernel
Hardware
① Formal Cap Distribution
② Security Architecture
⑤ Formal Security Property
④ Kernel Security Model

Progress



③ Trusted Component Behaviour
① Formal Cap Distribution
② Security Architecture
⑤ Formal Security Property
④ Kernel Security Model
Components
seL4 kernel
Hardware

Progress



③ Trusted Component Behaviour
Components
seL4 kernel
Hardware
① Formal Cap Distribution
② Security Architecture
⑤ Formal Security Property
④ Kernel Security Model

Conclusion

NICTA

- Full system verification of modern systems infeasible
  - But verification of specific, targeted properties feasible
- Presented a framework for proving security
  - Break code into components, avoid needing to trust the bulk of our functionality
  - Formally verify components capable of violating desired property
- Built SAC as a case-study
  - Uses seL4 microkernel as a secure foundation
  - Showed a model of the system is secure
- Ongoing work is to join security model with existing seL4 proof

Thank You