A Control Flow Integrity Based Trust Model

Ge Zhu

Akhilesh Tyagi

Iowa State University

Trust Model

- Many definitions of *trust.*
- Typically transaction level trust propagation/policy.
- Self-assessment of trust.
- A trust policy & security policy specification.
- Compiler level support for embedding security/trust policy monitoring.

Program level trust

- Traditional trust
  - Static (w.r.t. program, potentially dynamic w.r.t. information)
  - Transaction level
- Program level trust
  - Real-time
  - Program level

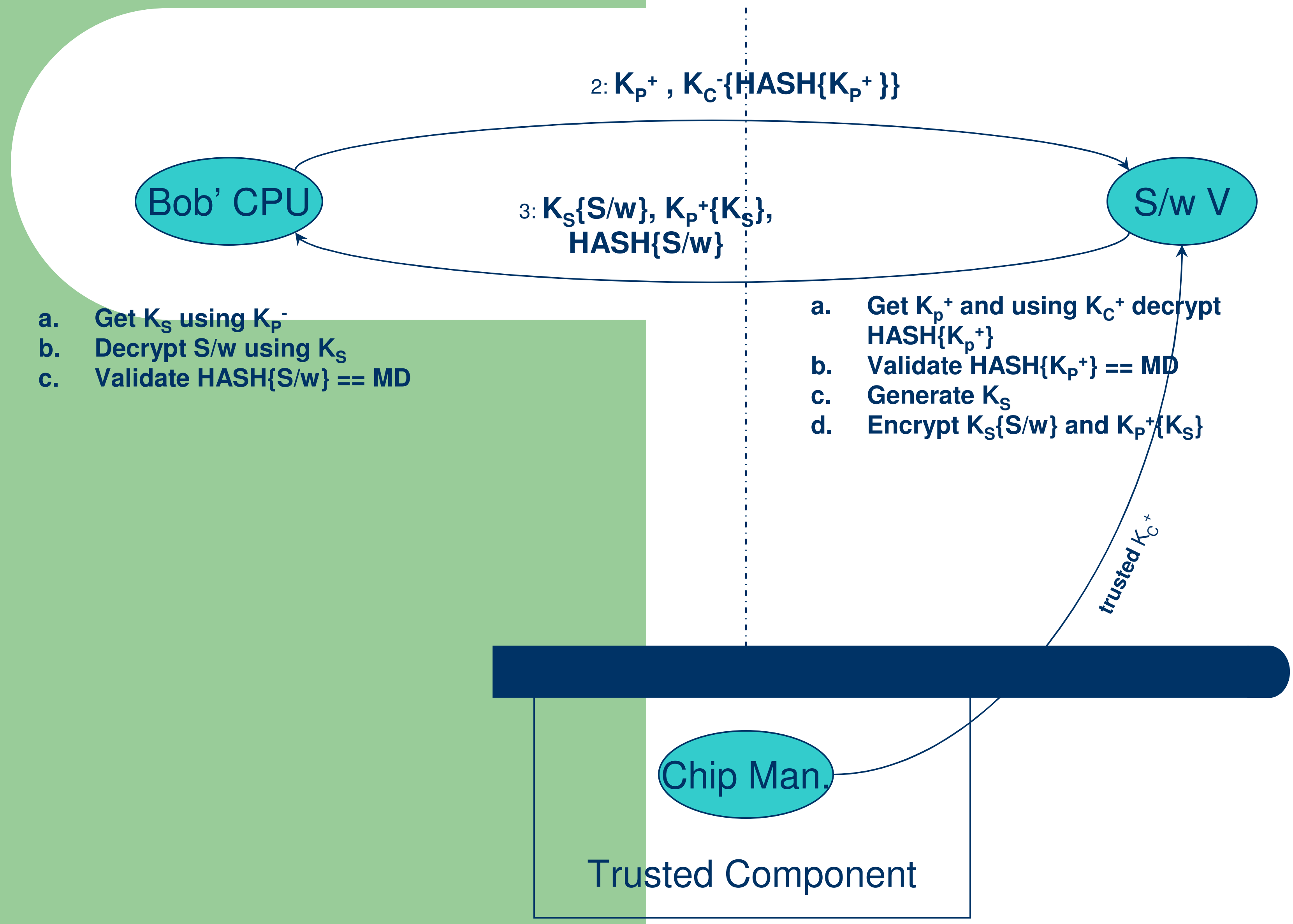
Architecture/Hardware Trust Support

- TCPA (TCG) Trusted Platform Module
  - Crypto co-processor (RSA -512, 768, 1024, 2048 bits; SHA-1; HMAC)
  - Components for asymmetric key generation, RNG, IO.
  - TPM may use symmetric encryption internally.
  - May implement other asymmetric components such as DSA or elliptic curve.
  - Endorsement keys/Attestation keys

Architecture/Hardware Trust Support

- TPM allows for a trust layer in a PDA, PC, Cell Phone.
- e.g. Integrity of the boot-up process.
- Allows for protection of intellectual property (keys, other data, programs).

Software distribution model



H/W System Level Trust

- Devdas *et al* use VLSI process variations to generate a signature of each hardware component.
- Develop a trust engine that composes system level trust?
- Trusted Circuits?

Back to Program Level Trust

- The underlying thesis is that control flow integrity of a program is a good indicator of its trustworthiness.
- Our hypothesis is that any program behavior compromise whether through data contamination or control contamination eventually is visible as control flow anomaly.

Basic scheme (cont.)

- We associate a dynamite trust level, a value in the range [0,1] with a subset of monitored entities in a program, which could be data structures or control flow edges.
- At runtime, the trust value will change according to embedded checks in the control flow.
- Trust here is an estimation of the likelihood of not breaching a given trust policy.

Control flow checking framework

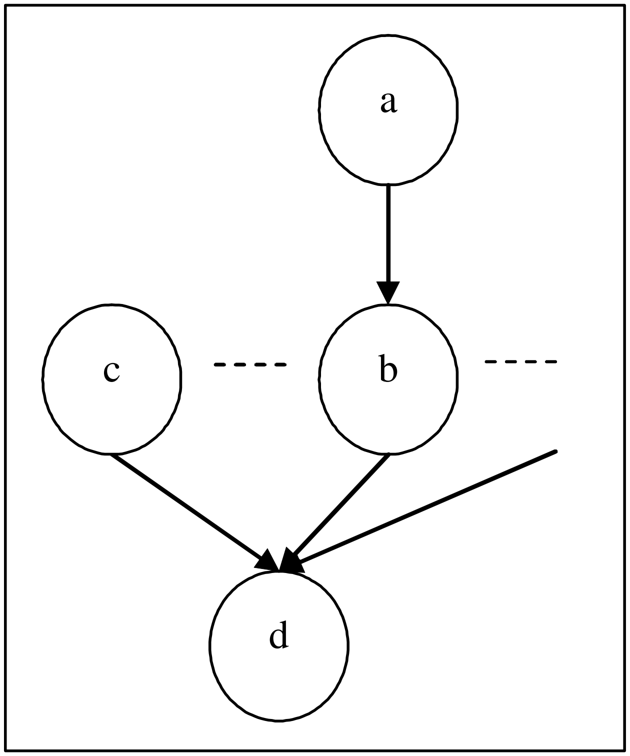
- McCluskey et al. proposed to use control flow signatures for fault tolerance in a processor.
- The signature model contains:
  - Each basic block *i* assigned a unique ID ID_i
  - Invariant: global register *GR* contains ID of the current block at exit.
  - Difference value for incoming edge *(j,i)* where j is the parent node for *i*, $D_{j,i} = ID_j \oplus ID_i$
  - Check for the consistency at *i*.

Travel over one edge

- Suppose control flow travels through *(a,b)*. At block *a*, we have $GR = ID_a$
- At block *b*, we need to check:

  $GR = GR \oplus D_{a,b}$

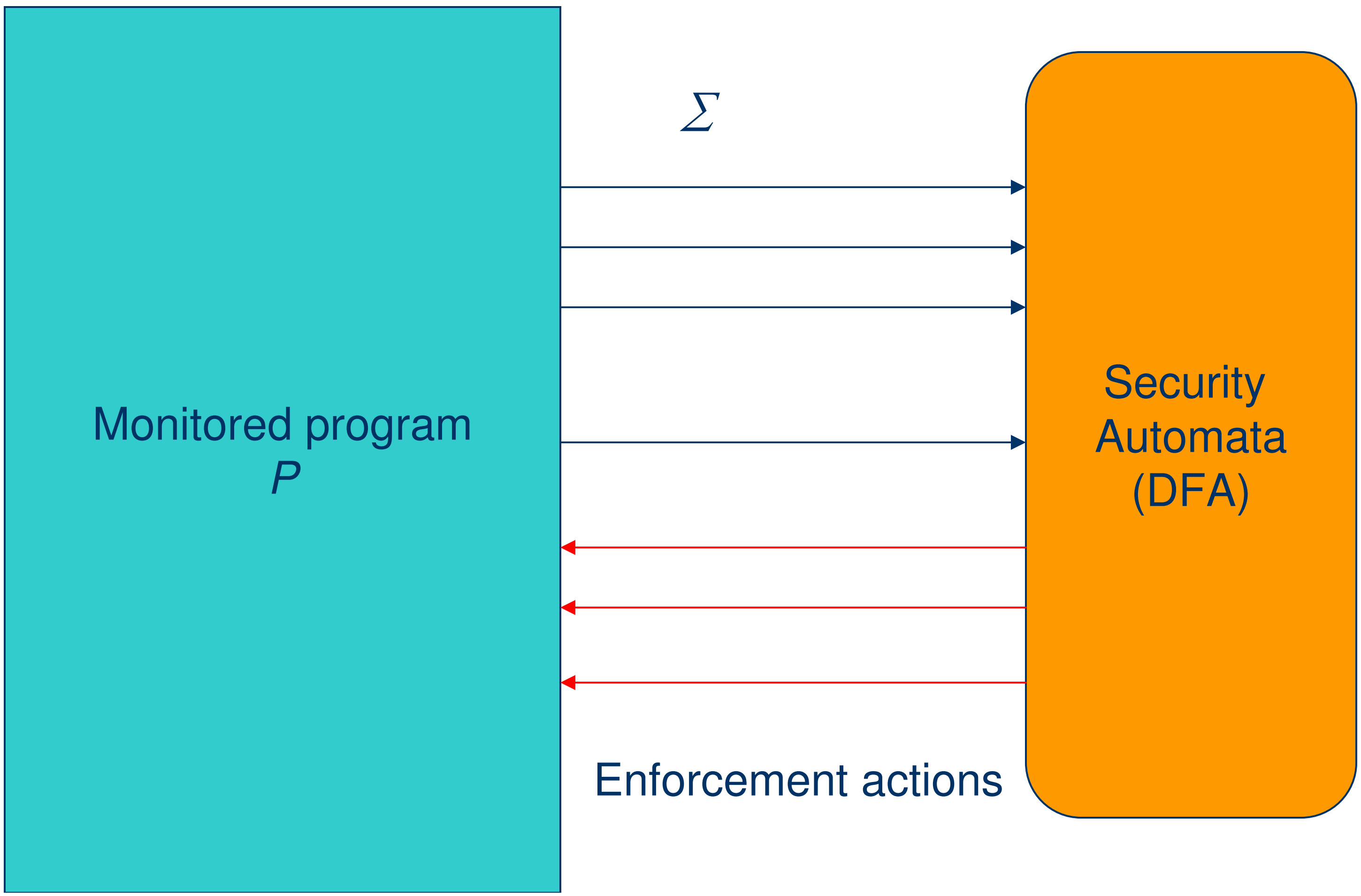
  $if\ (GR \neq ID_b)\ then\ \{\ error\}$



Control Flow Checking (CFC)`Framework

- The integrity of any subset of control flow edges can be dynamically monitored.
- Which ones should be monitored? How to specify these sets (ones that are monitorable)?
- Schnieder: security automata; Ligatti *et al*: Edit automata.

CFC Integrity Framework



CFC Integrity Framework

- A predefined set of monitored program events form Σ: each *malloc* call, access to the private key, buffer overflow – control flow edge after the procedure call return.
- What kind of finite sequences specify a *safety* property?
- *Security* and *edit* automaton.

Control flow checking automata

- An automaton is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_s, F)$$

where

Q is a finite set of states,
Σ is a finite set of symbols called the input alphabet,
δ is the transition function,
q_s is the initial state,
F is a finite set of final states.

CFC automata (cont.)

- A CFC automata is a security automaton which satisfies:

(1) $Q = (\bigcup_{a \in \Sigma} Q_a) \cup Q_s, \quad \text{where}$

$Q_s = \{q_s\}, \quad \text{and} \quad Q_{a|a \in \Sigma} = \{q \mid \delta(q', a) \rightarrow q\},$

$\text{and} \quad Q_{a|a \in \Sigma}, Q_s \quad \text{forms a set partition of } Q.$

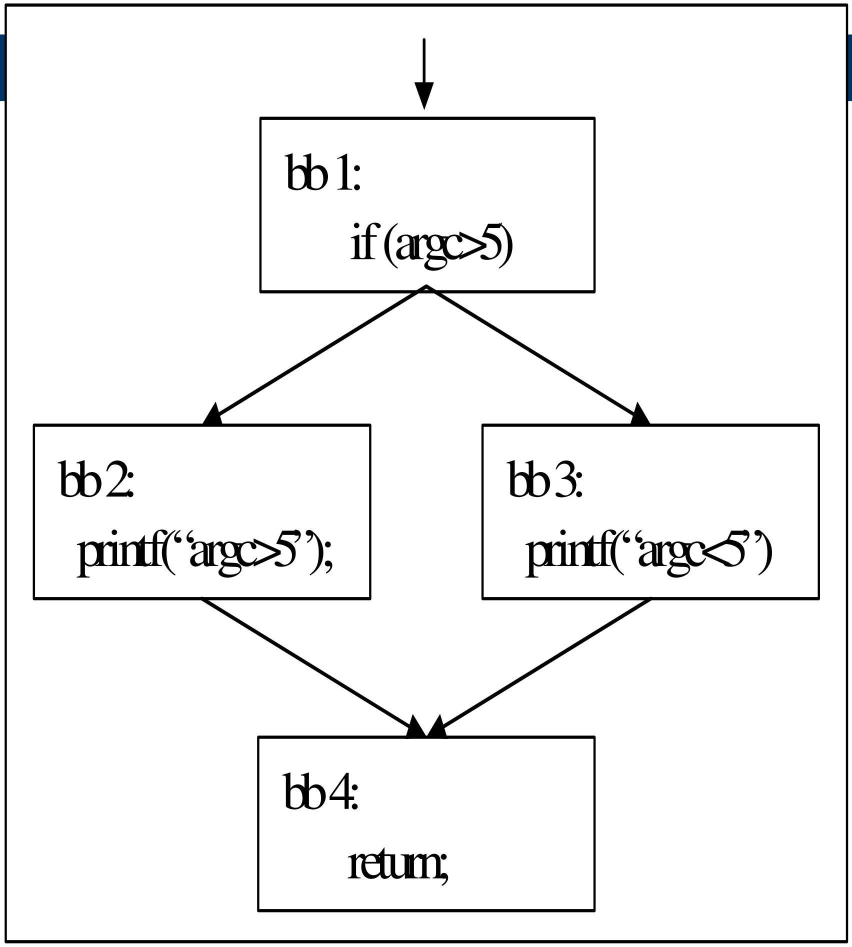
(2) $\neg(\exists q \in Q, \exists a \in \Sigma(\delta(q, a) \rightarrow q_s))$

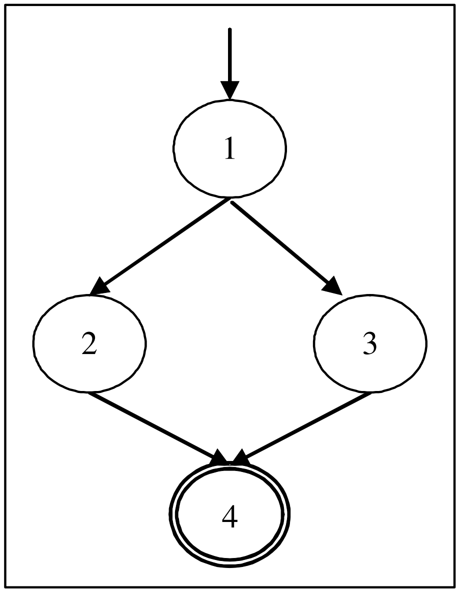
CFC DFA Example

- Build a control flow checking automaton for a simple program:

```
int main(int argc, char **argv){
    if (argc>5) { printf("argc>5\n"); }
    else { printf("argc<5\n"); };
    return;
}
```

Example (cont.)





Example (cont.)

- The CFC DFA is defined by $M = (Q, \Sigma, \delta, q_s, F)$ where:

$$Q = \{q_s, 1, 2, 3, 4\}$$

$$\Sigma = \{en_1, en_2, en_3, en_4\}$$

$$\delta = \{(q_s, en_1) \rightarrow 1, (1, en_2) \rightarrow 2, (1, en_3) \rightarrow 3,$$
$$(2, en_4) \rightarrow 4, (3, en_4) \rightarrow 4\}$$

$$F = \{4\}$$

- Notice that *en* is the event generated by control flow entering a new basic block.

Embed CFC automata into program

- The input to our algorithm would be a CFC DFA and a program *Prog* that needs to obey the security automaton. The output of our algorithm is a program *Prog'* with CFC DFA embedded into source code.
- We assume:
  - *P*: The set of program states
  - *Q*: The set of automaton states
  - *S*: The set of code insertion spots in the program

Embed (cont.)

$f_{QP} : Q \to P$

$$f_{QP} : Q \to P, \quad \text{where} \quad Q = (\bigcup_{a \in \Sigma} Q_a) \cup Q_s$$

is the predicate which maps automaton states into program states. We assume f_{QP} has the following two properties:

(1) For $q \in Q_a$ and $p \in Q_a$, $f_{QP}(q) = f_{QP}(p)$.

(2) For $q \in Q_a$ and $p \in Q_b$, $a \neq b \Rightarrow f_{QP}(q) \neq f_{QP}(p)$

Embed (cont.)

$f_{PS} : P \rightarrow S$

$f_{PS} : P \rightarrow S$ is the predicate which maps program states into code spots. For simplicity reason, we assume that :

(3) For $u, v \in P, u \neq v \Rightarrow f_{PS}(u) \neq f_{PS}(v)$

In complex situation, wehre (1) is not held, we could use conditional branch to decide what is the current program state at certain program spot.

Parent set

$Parent_q = \{q' \mid \delta(q', a) \rightarrow q\}$ is the parent set for $q \in Q$. And we have the following theorem.

Theorem 1: For a CFC DFA $M = (Q, \Sigma, \delta, q_s, F)$, where we have $p, q \in Q$ and $p \neq q$ and p, q are mapped into a same program spot, then we have

$$Parent_p \bigcap Parent_q = \varnothing$$

Theorem 1 proof

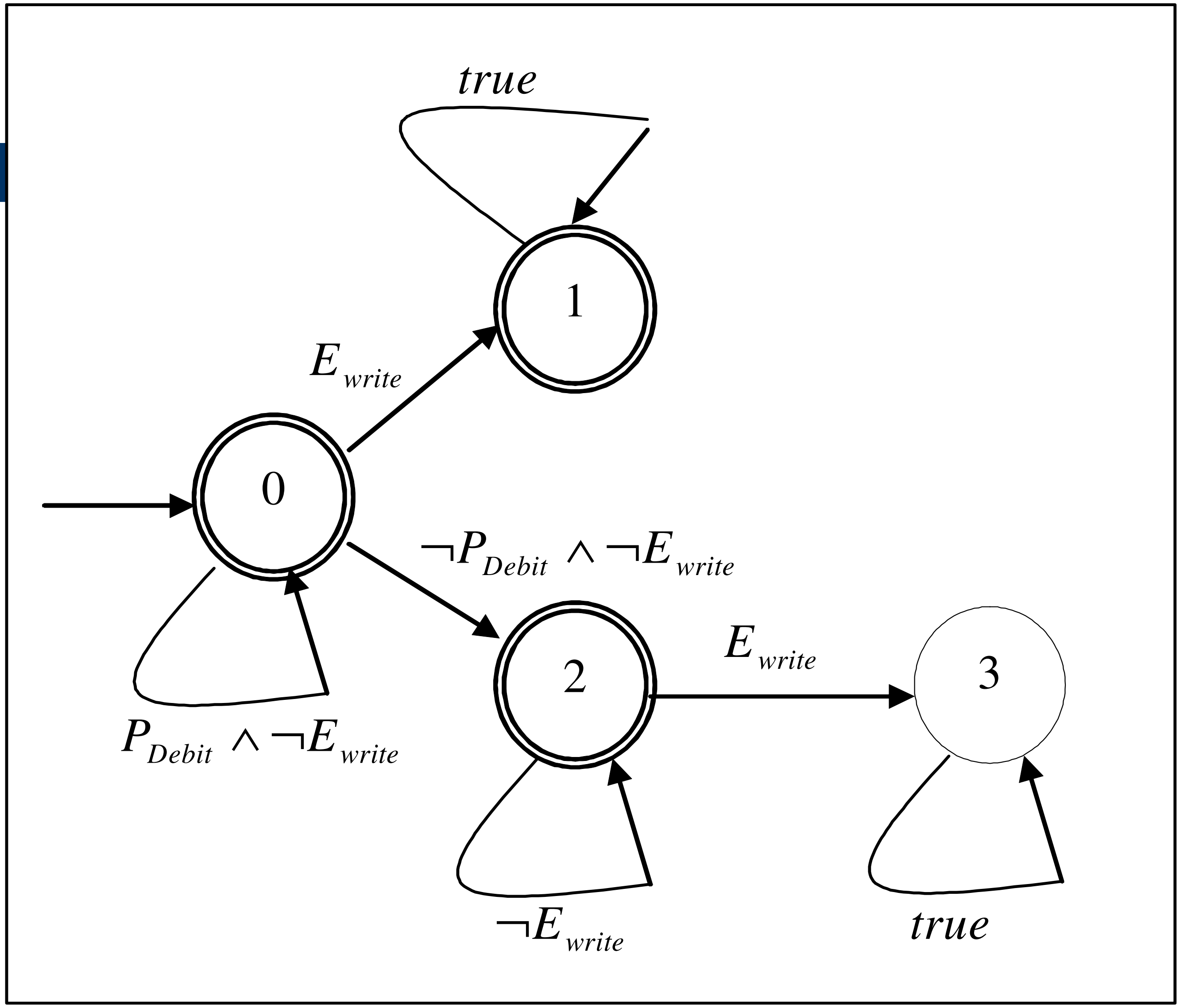
Suppose the contrary stands, i.e., $Parent_p \bigcap Parent_q \neq \emptyset$, and we assume that $r \in Parent_p \bigcap Parent_q$.

Assume that $u = f_{QP}(p)$ and $v = f_{QP}(q)$. As we know, $f_{PS}(u) = f_{PS}(v)$. From (3), we know that $u = v$, i.e., $f_{QP}(p) = f_{QP}(q)$. From (1), (2), we know that there exists $a \in \sum : p \in Q_a, q \in Q_a$. As $r \in Parent_p \bigcap Parent_q$, we then know $\delta(r, a) \rightarrow q$ and $\delta(r, a) \rightarrow p$. This contradicts with the fact that M is a $CFC\ DFA$. Done.

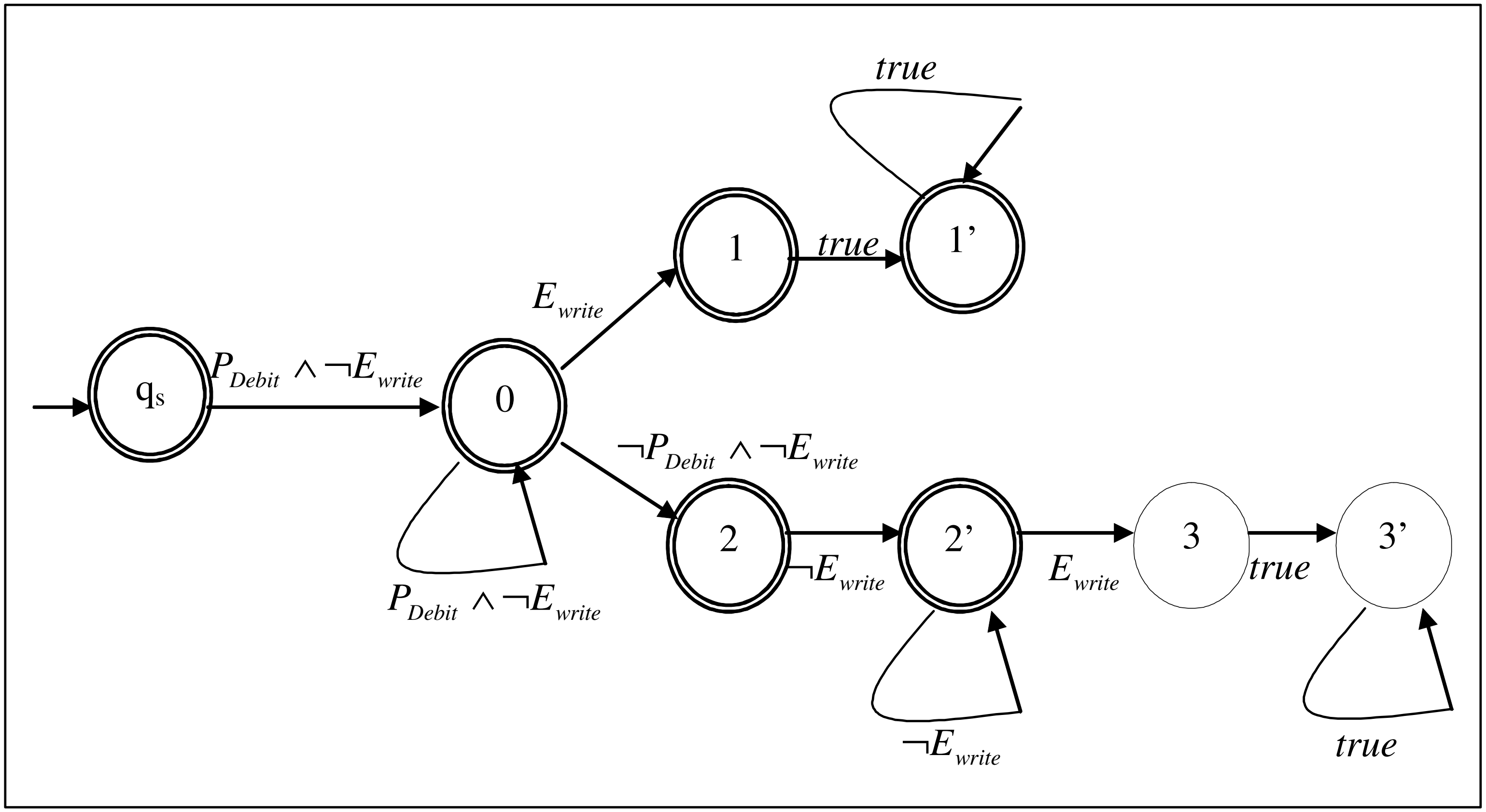
Example 1

- Electronic commerce example (*F. Bession et al., "Model checking security properties of control flow graphs"*)
- The security automaton ensures that either there are *no writes* or all the codes leading to *write* have *Debit* permission.
- *Ewrite* stands for the action of write.
- *Pdebit* stands for the permission to debit.

Example 1 (cont.)



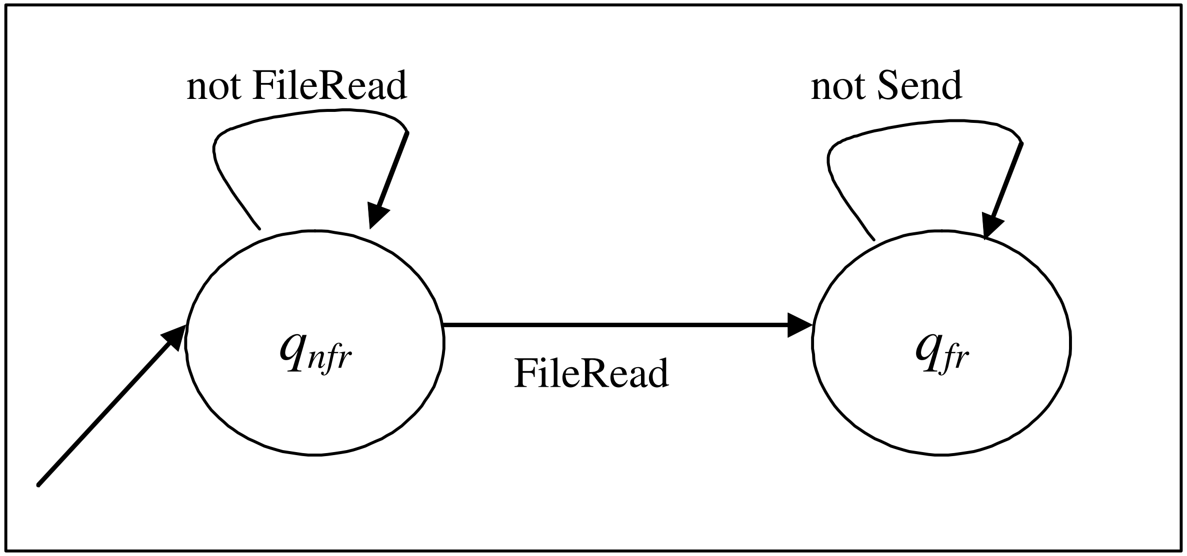
Example 1 (cont.)



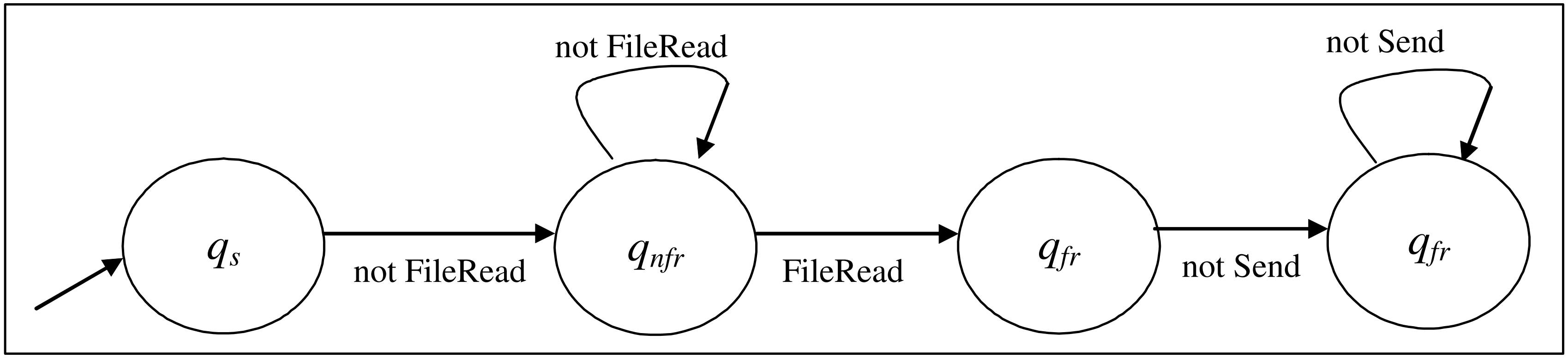
Example 2

- F. Schneider, "*Enforceable security policies*"
- The following security automaton specifies that there can be no *send* action after a *file read* action has been performed.

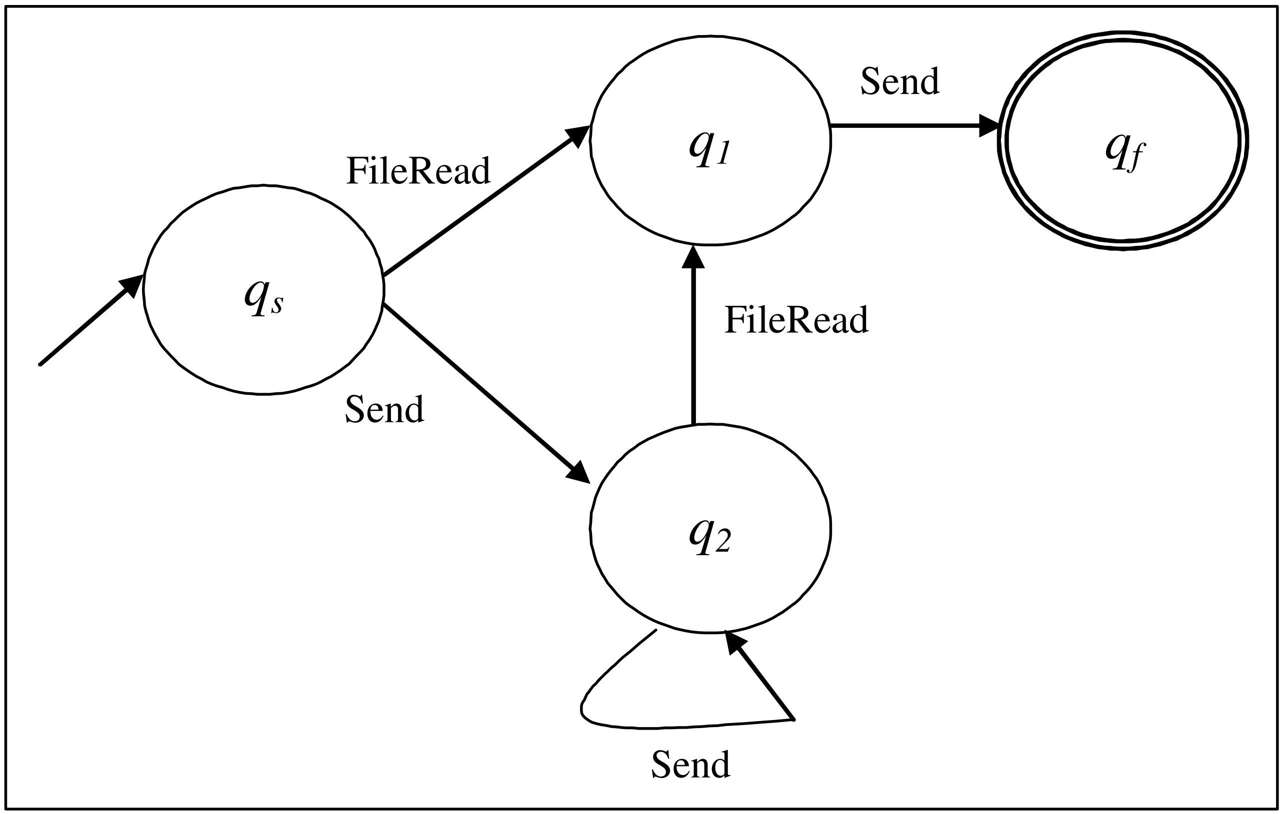
Example 2 (cont.)



Example 2 (cont.)



Example 2 (cont.)



Trust Policy

- We view *trust* with respect to a specified security policy.
- If a security policy is violated, trust w.r.t. that attribute is lowered.
- Trust policy just an enhancement of security policy accounting for updates of the trust value.

Trust Automaton

- Trust automaton: $$M = (Q, \Sigma, \delta, t, q_s, F)$$
- *t* is the trust update function: *t(q,a) = val*
- Could be a multi-dimensional update.
- When trust is lowered below a certain threshold, an exception could be raised.
- Exception could call an appropriate service such as *intrusion detection system* or *trust authentication service*.

Experimental results

- We have compiled and run two of the SPEC2000 benchmarks *gzip* and *mcf* to evaluate both static and dynamic system overhead.

Experimental results (cont.)

- Static system overhead

Program	Old blocks	New blocks	Increased	Old Insns	New Insns	%Increase
gzip	1730	3945	128.03%	17429	73047	319.11%
mcf	395	962	143.54%	4565	17937	292.92%

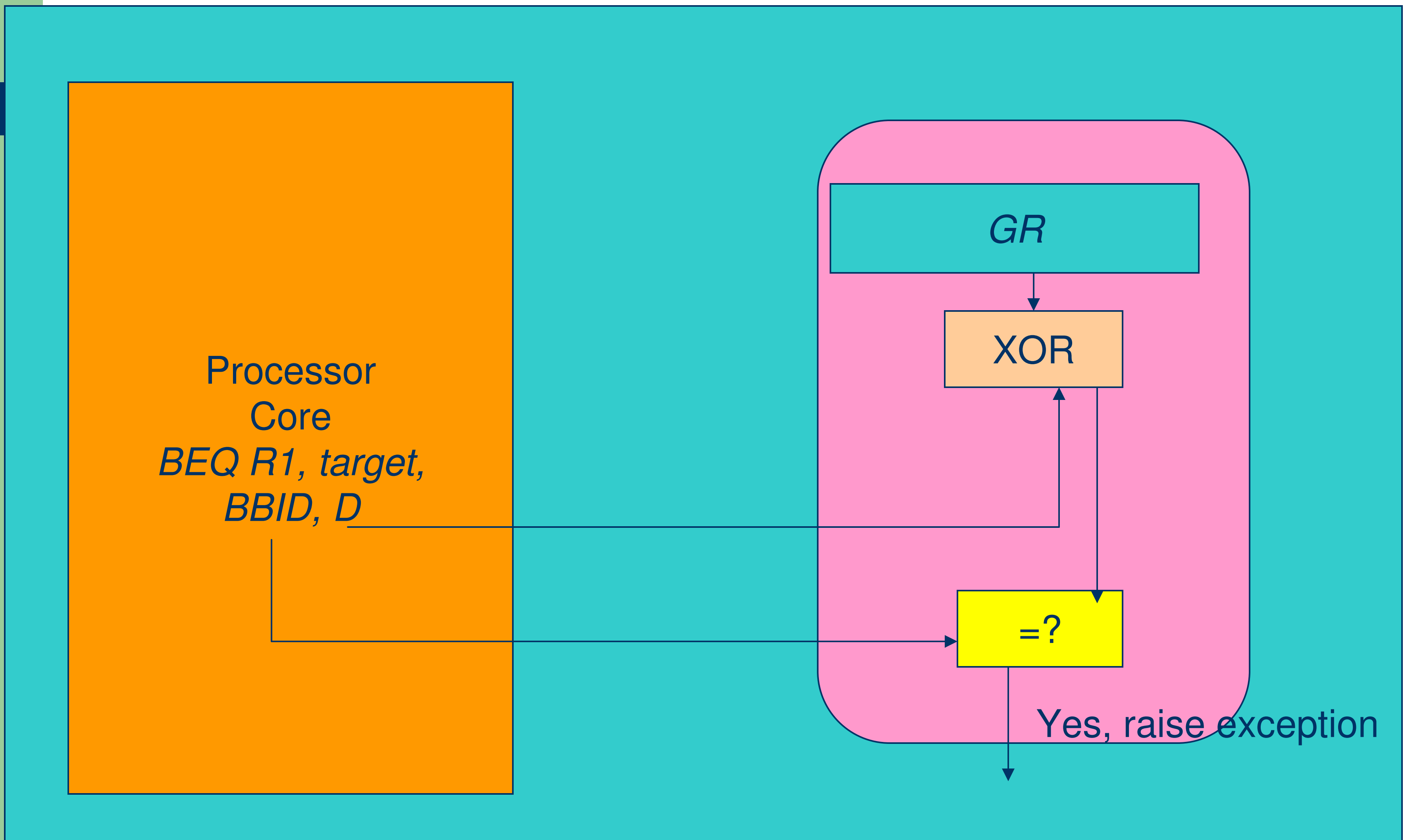
- Dynamic system overhead

Program	Number of dynamic checks (billion)	Reference Time	Base Runtime	Base ratio
164.gzip	128	1400	11969	11.7
181.mcf	22	1800	1611	112

Architecture Level support

- The performance overhead will be significantly reduced if the architecture manages the trust attributes.
  - Associate extra attributes with *branch* instructions:
  - `BEQ R1, target, BBID, D`
  - Being implemented in SimpleScalar.

Trust Engine Based processor



Conclusions

- We proposed a control flow integrity based trust model.
- program's self assessment of trust.
- compiler driven approach.
- performance overhead.
- Trust engine based architecture for higher efficiency.