An Online Finite LTL Model Checker for Distributed Systems

-- with an introduction to our group

Zhengwei QI

May 20, 2011

Shanghai Jiao Tong University –Shanghai, China

Current SCS visiting faculty –hosted by Prof. Ed Clarke

Agenda

- Introduction
- **An Online Model Checker for Distributed Systems**
- A Refined Decompiler to Generate C/C++ Code with High Readability
- Current research topics in our group

About Me

- 2002-2006
  - PhD, SJTU, my supervisor is Prof. Jinyuan YOU
- 2006-present
  - Teacher, School of Software, Shanghai Jiao Tong University
- 2008.1-2008.7
  - a visiting teacher in the **system group** of Microsoft Research Asia

About Shanghai Jiao Tong University (SJTU)

- 1896, Nanyang Public School in Shanghai
- Today
  - 26 academic schools and departments
  - 63 undergraduate programs
  - 232 masters-degree programs
  - 147 Ph.D. programs
  - 20,300 undergraduates,
  - 28,100 masters and Ph.D. candidates
  - 1,900 professors and associate professors

About School of Software

- 2001.5-2011.5
  - Celebrations for the 10th anniversary of the establishment
- Today
  - 150 new undergraduates per year
  - 60 new masters per year
  - 15 new PhD candidates per year
  - 30+ faculty
  - 9 labs including System\Software engineering\Applications

Agenda

- Introduction
- **An Online Model Checker for Distributed Systems [with Microsoft Research Asia]**
- A Refined Decompiler to Generate C/C++ Code with High Readability
- Some research topics in our group

Debugging distributed systems is difficult

- Bugs are difficult to reproduce
  - Many machines executing concurrently
  - Machines may fail
  - Network may fail
- Existing Methods
  - Insert print statements, then writes a front client to pars
  - model checkers find safety counterexamples
  - software model checking methods are focused on the specifications (Spin/SMV/TLC)

Microsoft Research Asia
System Research Group

Example: Paxos protocol [Leslie Lamport]

Application: Microsoft Autopilot cluster management service
Google Chubby distributed lock service

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  (1)
   |         X--------->|->|->|       |  |  (2)
   |         |<---------X--X--X       |  |  (3)
   |         X--------->|->|->|       |  |  (4)
   |         |<---------X--X--X------>|->|  (5)
   |<---------------------------------X--X  (6)
   |         |          |  |  |       |  |
(1):Request  (2):Prepare(N)
(3):Promise(N,{Va,Vb,Vc})
(4):Accept!(N,Vn)  (5):Accepted(N,Vn)
(6):Response
```

Liveness property :
Some proposed value is eventually chosen and, and if a value has been chosen, then a process can eventually learn the value.

Microsoft Research Asia
System Research Group

Problems for State-of-the-art of runtime checking

Step 1: add logs

```
void ClientNode::OnLockAcquired(…) {
    …
    print_log( m_NodeID, lock, mode);
}
```

Step 2: Collect logs, align them into a globally consistent sequence

- Keep partial order

Step 3: Write checking scripts

- Scan the logs to retrieve lock states
- Check the consistency of locks

- Too many manual effort
- Only low-level safety properties

Microsoft Research Asia
System Research Group

Problems for software model checking methods

- The real system code is complex usually, so it is not practical to use classical model checking to verify it which often leads to infinite states.
- Although some errors can be found by checking the abstract model, many bugs related to the real code are still hard to be detected
- There are many optimization tricks undefined in specification to improve performance, which increases difficulty for evaluating their side-effects.
- *Our focus*: provide online dynamic monitoring tool to check whether a real system satisfies a set of high-level safety and liveness properties.

Microsoft Research Asia
System Research Group

Why LTL?

- "Modern software model checkers find *safety violations: breaches* where the system enters some bad state. However, we argue that checking *liveness properties offers both a richer and more natural* way to search for errors, particularly in complex concurrent and distributed systems". [NSDI 2007]

  Liveness properties specify desirable system behaviors which must be satisfied *eventually, but are not always satisfied, perhaps as a result of failure or during system* initialization.

Classical Linear Temporal Logic and Finite Trace Semantics

Definition *(syntax). The set of LTL formulae on the set* $\mathcal{P}$ *is defined by the grammar* $\varphi ::= p|\neg\varphi|\varphi \vee \varphi| \bigcirc \varphi|\varphi U \varphi$, *where* p *ranges over* $\mathcal{P}$.

Definition *(semantics) The satisfaction relation of Finite Trace Linear Temporal Logic (FLTL)* $\models \subseteq Trace \times Formula$ *defines when a trace* t *satisfies a formula* f, *written* $t \models f$, *and is defined inductively over the structure of the formulae as follows, where A is any atomic proposition and X and Y are any formulae:*

Finite Trace Semantics

$$
\begin{array}{llll}
t \models true & iff & true & (1) \\
t \models false & iff & false & (2) \\
t \models A & iff & A \in head(t) & (3) \\
t \models \neg A & iff & t \not\models A & (4) \\
t \models X \wedge Y & iff & t \models X \quad and \quad t \models Y & (5) \\
t \models X \vee Y & iff & t \models X \quad or \quad t \models Y & (6) \\
t \models \bigcirc X & iff & if(tail(t) = \emptyset) \quad t \models X & \\
 & & else \quad tail(t) \models X & (7) \\
t \models \Diamond X & iff & (\exists i \leq length(t)) \quad t_i \models X & (8) \\
t \models \Box X & iff & (\forall i \leq length(t)) \quad t_i \models X & (9) \\
t \models X \; U \; Y & iff & (\exists i \leq length(t)) \quad (t_i \models Y & \\
 & & and \; (\forall j < i) \; t_j \models X) & (10)
\end{array}
$$

It is acceptable to regard a finite trace as an infinite stationary trace in which the last event is repeated infinitely [Grigore Rosu, et al. 2005]

Modified Büchi automaton

- Our automata $A(\varphi) = (S, \Sigma, S_0, \delta, F)$
  - S is *the set of states*
  - Σ *is the alphabet*
  - S_0 *is the initial set*
  - δ *is transition relation*
  - *F is the accepting condition.* $F = \{F_1, F_2, \ldots, F_n\}$, *Because* eventualities must be satisfied on the finite sequence, so the accepting condition is

$$f \in F, iff \forall i, 1 \leq i \leq n, f \in F_i$$

The correctness proof

Theorem *Let P be the set of propositions from which LTL formulas are constructed, $\xi = x_0, x_1, \ldots, x_n$ is a finite word over 2^P, and $A(\varphi)$ is the finite automaton for formula φ. Then $A(\varphi)$ accepts ξ iff $\xi \models \varphi$.*

- Notations and proof, from [Rob Gerth, et al, PSTV 1995]
  - *Old*(*s*) denote the set of formulas that must hold and have already been processed in node *s*
  - *New*(*s*) denote the set of formulas that must hold at current state and have not yet been processed in *s*
  - *Next*(*s*) denote the set of formulas that must hold in all immediate successors of *s*
  - Δ(*s*) denote the value of *Old*(*s*) when the construction of *s* is finished.

Lemma 1

- *Lemma 1* For every initial state $q \in I$ of an automaton A generated from the formula φ, we have $\varphi \in \Delta(q)$.
- Proof. Immediately form the construction. ■

Lemma 2

- *Lemma 2* Let σ = $q_0q_1q_2$... be a run of A that accepts the propositional sequence ξ when q_0 is taken to be an initial state. Then

$$\xi \models \wedge\Delta(q_0)$$

- Proof sketch. By induction on the size of the formulas.
  - The base case is for formulas of the form P, ¬P.
  - We show the case of $\mu U \eta \in \Delta(q_0)$ according to the construction of U operator, only following two cases are possible:

$$1. \forall i \geq 0 : \mu, \mu U \eta \in \Delta(q_i) and \eta \notin \Delta(q_i)$$

$$2. \exists j > 0 \forall 0 \leq i < j : \mu, \mu U \eta \in \Delta(q_i) and \eta \in \Delta(q_j)$$

  - Since σ satisfies the acceptance conditions of A, only case 2 is possible. By the induction hypothesis, then $\xi_j \models \eta$ and $0 \leq i \prec j, \xi_i \models \mu$, then $\xi \models \mu U \eta$

■

Lemma 3

- Lemma 3 Let σ be an execution of the automaton A, constructed for φ, that accepts the propositional sequence ξ. Then $\xi \models\Rightarrow \varphi$
- Proof.
  - The node q_0 is initial state, From Lemma 2 it follows $\xi \models\Rightarrow \wedge\Delta(q_0)$
  - By lemma 1, if q_0 is initial then $\varphi \in \Delta(q_0)$
  - Thus, $\xi \models\Rightarrow \varphi$ ∎

Lemma 4

- Lemma 4 If $\xi \models \varphi$, Then there exists an execution σ of A that accepts ξ.
- Proof sketch.
  - First, there exists a node that q_0 is initial such that
    $$\xi \models (\wedge\Delta(q_0)) \wedge X(\wedge Next(q_0))$$
  - Now if $\xi_i \models (\wedge\Delta(q_i)) \wedge X(\wedge Next(q_i))$,according to *transition invariants* of automaton, we can find a successor q_{i+1} of q_i that $\xi_{i+1} \models (\wedge\Delta(q_{i+1})) \wedge X(\wedge Next(q_{i+1}))$
  - Since $\xi_i \models \mu U \eta$, there must be some minimal $j \geq i$ such that $\xi_j \models \eta$. ∎

Architecture

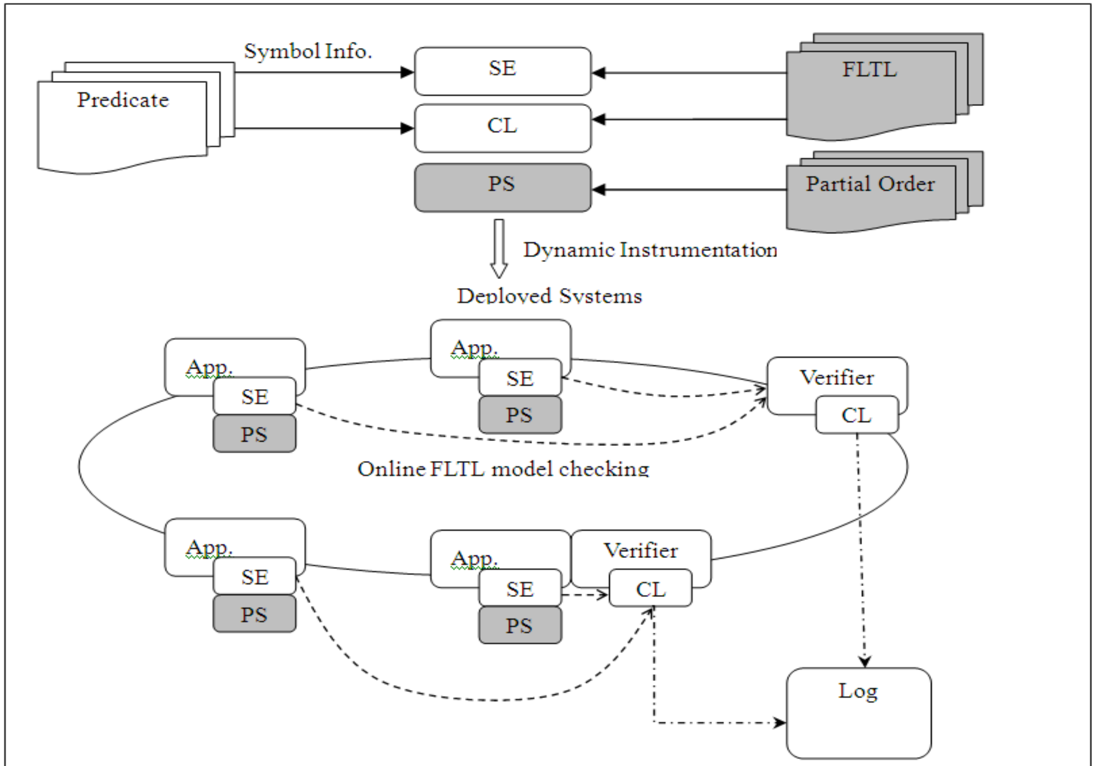
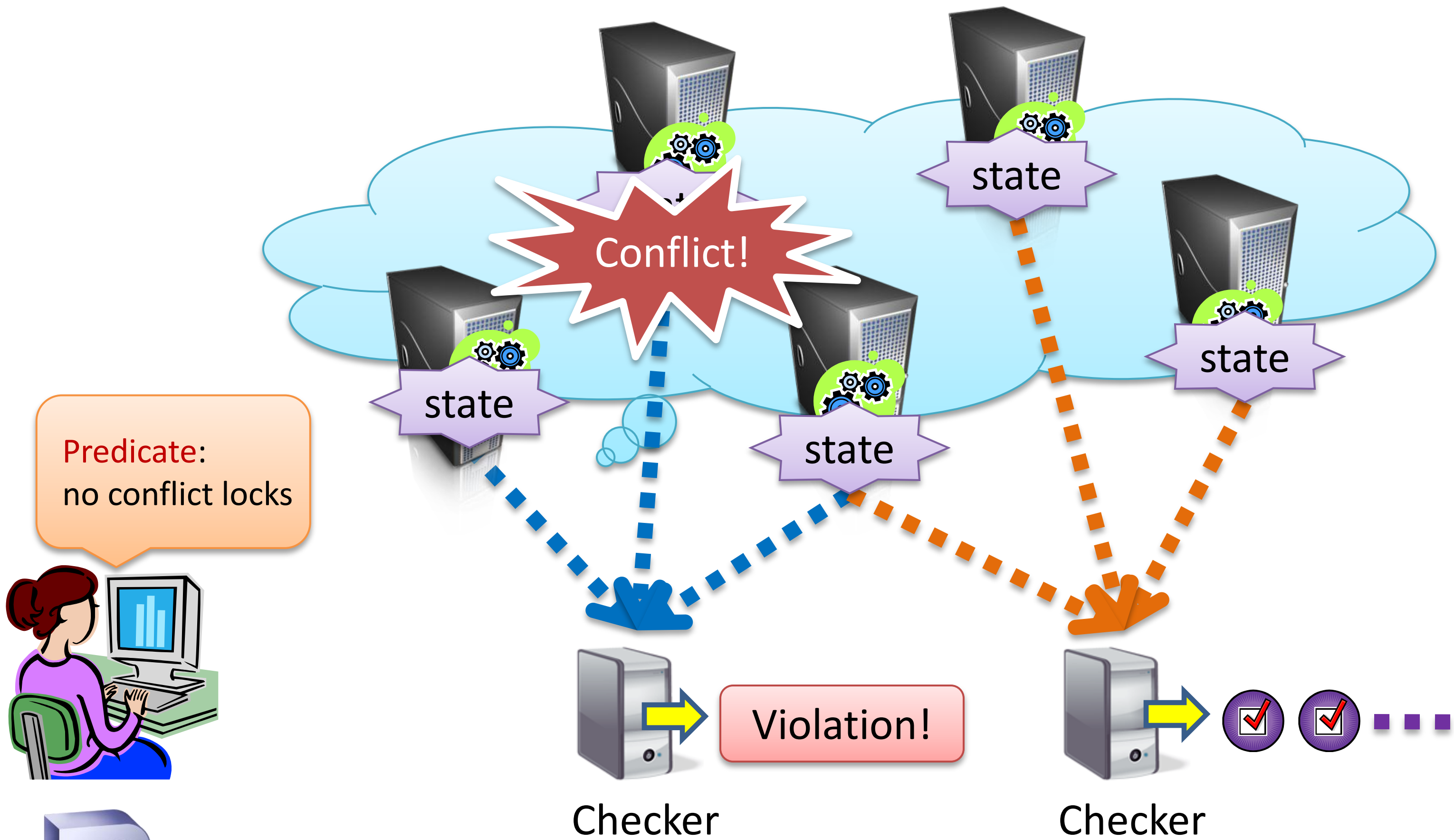


Fig.1 Architecture

- State Exposer(SE)
  - uses MSRA's tool D³S [NSDI 2008] to instrument processes being monitored
  - SE loads the DLL into the process's address space, and redirects function calls that are interposed on to callbacks in the DLL.
- Verifiers
  - collect states that are transmitted from SE, then evaluate predicates and output bug reports.
  - the modified version of the popular algorithm SPIN [Gerard J. Holzmann, 1997]

Microsoft Research Asia
System Research Group

D^{3}S Workflow [From NSDI 08]



Microsoft Research Asia
System Research Group

D3S Interface-1

New Class derived from D^3S

```
• class exposer : public beyond::d3s::emit::actor_io_service<exposer,fltl_exposer>
{
public:
    static void     execute(const state & param) {
            std::cout << "[exposer_actor_io_service] " << &param << std::endl;
            beyond::d3s::emit::emit_to<fltl_partitioner>(param);     }
    ……

};
```

Microsoft Research Asia
System Research Group

D3S Interface-2

1. Obtain the distributed states

- class exposer : public beyond::d3s::emit::actor_io_service<exposer,fltl_exposer>

2. Ordered states according to happen before relation

- class partitioner : public beyond::d3s::emit::actor_partitioned_call_through<partitioner,fltl_partitioner>

3. Call modified SPIN engine to check FLTL

- class verifier : public beyond::d3s::emit::actor_sorter_called_in_io_service<verifier, fltl_verifier>

Microsoft Research Asia
System Research Group

Modified SPIN

```
1 if (incr_cnt+count >= Max_Red)
2    sprintf(pref,"accept");/*last hop*/
3 else
4 //sprintf(pref,"T%d",count+incr_cnt);
5    sprintf(pref, "T0");
```

- Line 4: if not all the right part of U operator are implemented in this state, then go to the T(count+incr cnt) level to continue
- In case of finite trace
  - the last step should be repeated infinitely, so all the right part of U operator should be satisfied in the last step.
  - If not, we just go to the T0 level to reject this formula(Line 5)

$$\Box(propose \rightarrow \Diamond chosen)$$

```
T0_init:
  if
  :: ((! ((propose)) || (chosen))) -> goto
       accept_S20
  :: (1) -> goto T0_S27
  fi;
accept_S20:
  if
  :: ((! ((propose)) || (chosen))) -> goto
       T0_init
  :: (1) -> goto T0_S27
  fi;
accept_S27:
  if
  :: ((chosen)) -> goto T0_init
  :: (1) -> goto T0_S27
  fi;
T0_S27:
 if
 :: ((chosen)) -> goto accept_S20
 :: (1) -> goto T0_S27
 :: ((chosen)) -> goto accept_S27
 fi;
}
```

Online Program Analysis

```
CurrentStateList currentstate={InitState}
NextStateList nextstate={}
CheckOneStep(stateformula, finalstep){
  result = NoProgress
  foreach state in currentstate {
    foreach transition in state {
      if(stateformula == transition.
          Condition){
       nextstate.Add(transition.TargetState
          )
       if(finalstep){
         if(transition.TargetState.accept){
           return Accept
         }
         continue
       }
       result = Progress
      }
    }// end for each transition
  }// end for each state

  currentstate.clear()
  currentstate = nextstate
  nextstate.clear()

  if(finalstep && result != Accept){
    return Reject
  }
  return result
}
```

Monitor per step

If it's final step, Check accept condition

Microsoft Research Asia
System Research Group

Case study- Paxos

- Paxos protocal [Leslie Lamport] - Concurrent , distributed state machine for **Consensus**
- Three main state:
  - Stable

    In this state R believes it knows all chosen decrees, and it has accepted exactly one additional decree which may or may not have been chosen.
  - Initializing

    R starts in this state after replaying its log. It also enters the initializing state whenever it receives a message which shows that a decree has been chosen which R has not heard about, or when no decrees have been passed for a while.
  - Preparing

    In this state R is trying to elect itself primary. It sends Prepare requests to all peers, and if a majority responds R moves to the Stable state as primary.

Microsoft Research Asia
System Research Group

Safety

- Nontriviality: all the learned value must be the proposed value

Definition 5 $p1 \triangleq \forall r \in Learner : (learned[r] \in proposedset)$

$$Nontriviality \triangleq \Box p1$$

- Stablity: when a value is learned, this value will always be learned.

Definition 6 $p2 \triangleq \forall r \in Learner : (learned[r] = v)$
$p3 \triangleq \forall r \in Learner : (learned[r] \subseteq v)$
$Stability \triangleq \Box(p2 \rightarrow \Box p3)$

- Consistency: any two replica will learn the same value.

Definition 7 $p4 \triangleq \forall r1, r2 \in Learner : (learned[r1] = learned[r2])$
$Consistency \triangleq \Box(p4)$

Microsoft Research Asia
System Research Group

Liveness

- "We won't try to specify precise liveness requirements. However, the goal is to ensure that some proposed value is eventually chosen and, if a value has been chosen, then a process can eventually learn the value."[Leslie Lamport,2001]

Definition 8 $p5 \triangleq (chosen \vee notchosen)$
$Progress1 \triangleq \Box(propose \rightarrow \Diamond p5)$

Definition 9 $p6 \triangleq (chosen \wedge requirevalue)$
$Progress2 \triangleq \Box(p6 \rightarrow \Diamond learned)$

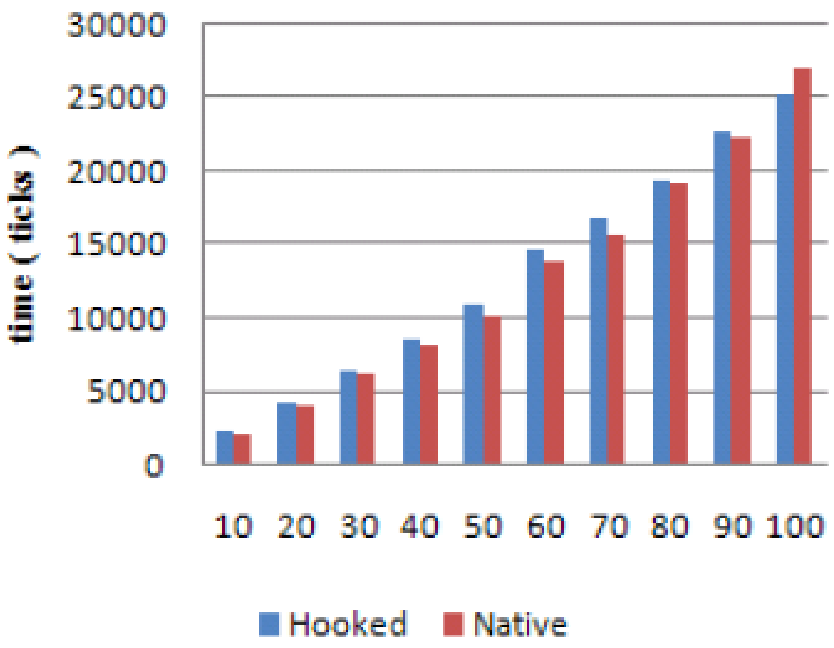
Microsoft Research Asia
System Research Group

Program Analysis Challenges

- How to define a global state and expose it as the state predicate ?
  - the global state can be defined as the array of tuple (ReplicaID, State, Ballot, Decree, Value) with logical timestamp.
- How to specify the final step?
  - if all replicas have accepted a consistent decree, we know that previous round must be ended and then indicate that previous round reaches its final step
- How to go through the different paths as many as possible?
- design three execution
  - models: Message Model, Restart Model, and Reconfig Model.
  - Future: Use some model checker to cover different paths.

Microsoft Research Asia
System Research Group

Experiment Evaluation



From this result, the overhead is less than %5 in most cases.

Microsoft Research Asia
System Research Group

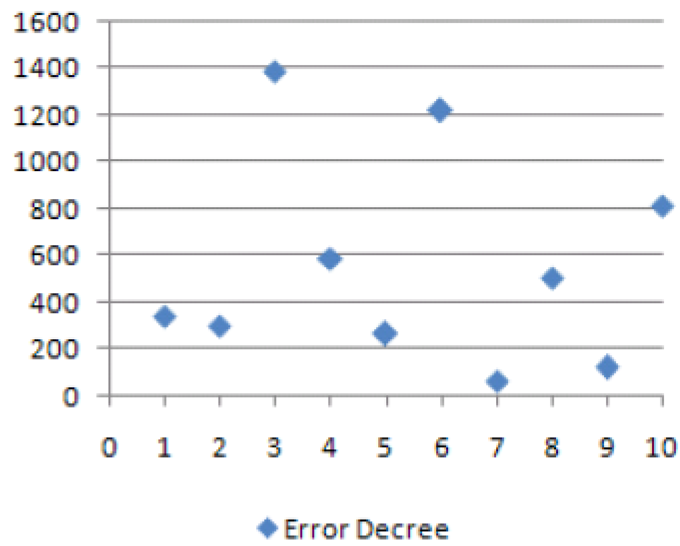
Experiment Evaluation-2

Table 1. Property Checking Result

Model/Property	Message	Restart	Reconfig
Nontriviality	√	√	√
Stability	√	√	×
Consistency	√	×	×
Progress1	×	×	×
Progress2	√	×	×

- Typical Bug: When Replica 4 learned Decree 368 in Ballot 41 and executed this request, we found that the previous decree (Decree 367) had not been learned while other replicas all learned Decree 367, which violates Consistency and Progress1 properties.
- This bug validates the high-level properties and involves several rounds which can not be easily captured by simple predicates such as assert().

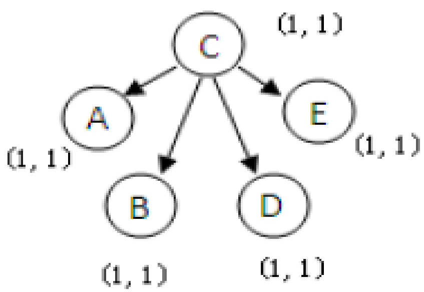
Experiment Evaluation-3



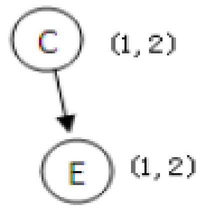
- We may find bugs in 2000 rounds and take less than one hour.
- After fixing these bugs we run our tool again, and have not found bugs in 4000 rounds.

Microsoft Research Asia
System Research Group

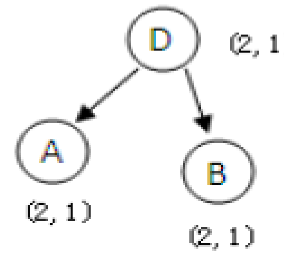
A Livelock bug



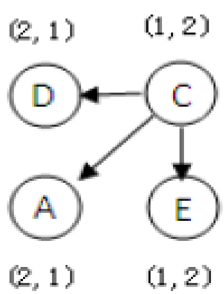
(1) Node C proposed a value to A,B,D,E



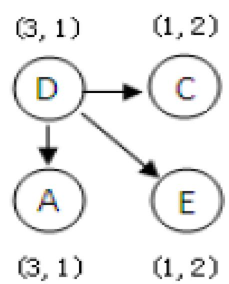
(2) C proposed a value to all, but only E accepted



(3) A,B,D restarted and D became the proposer



(4) C,E restarted and C began to propose its last value again



(5) D began to compete for being proposer, and live-lock started

Related Work

- Check safety properties in systems
  - Using random walks to analyze networking protocols whose state spaces were too large for exhaustive search [PSTV'86].
  - A method for iterating exhaustive search and random walks to find bugs in cache-coherence protocols.[PDMC'03]
  - WiDS and D^3S [NSDI'07, NSDI'08]

- Model checking software implementations is to abstract them to obtain a finite-state model of the program
  - Verisoft [POPL'97]
  - CMC [OSDI'01]
  - JavaPathfinder[TACAS '04]
  - SLAM [POPL'02]
  - SAT-solver[TACAS'04]
  - CUTE/CREST [PLDI'08 FSE'08 CAV'06]
  - Eagle/ JMPAX [FMOODS'05]

- we provide a high level temporal logical description to find safety and liveness violations in real code

Microsoft Research Asia
System Research Group

Agenda

- Introduction
- An Online Model Checker for Distributed Systems [with Microsoft Research Asia]
- A Refined Decompiler to Generate C/C++ Code with High Readability [WCRE 2010]
- Some research topics in our group

Motivation

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR argv[])
{
  int a[2][3] = {1,2,3,4,5,6};
  int b[3][2] = {1,2,3,4,5,6};
  int c[2][2] = {0,0,0,0};

  for(int i = 0; i < 2; i++)
    for(int k = 0; k < 2; k++)
      for(int j = 0; j < 3; j++){
          c[i][k] += a[i][j]*b[j][k];
      }
}
```

(a) Source Code of MatrixMul

```
extern char aS_0[41];
extern _UNKNOWN sub_418C30;
extern _UNKNOWN sub_4197E0;
extern _UNKNOWN loc_41983C;
...

_DWORD __cdecl sub_41100A(_DWORD, _DWORD,
_DWORD, char);
_DWORD sub_41100F();
_DWORD __cdecl sub_411023(UINT CodePage);
_DWORD __cdecl sub_41102D(_DWORD);
...

if ( *(_DWORD *)(v7 + v3 - 4) != -858993460 || *(_DWORD
*)(v7 + *(_DWORD *)(v6 + 4) + v3) != -858993460 )
 sub_41125D( r, *(_DWORD *)(*(v5 + 1) + v4 + 8));
...
```

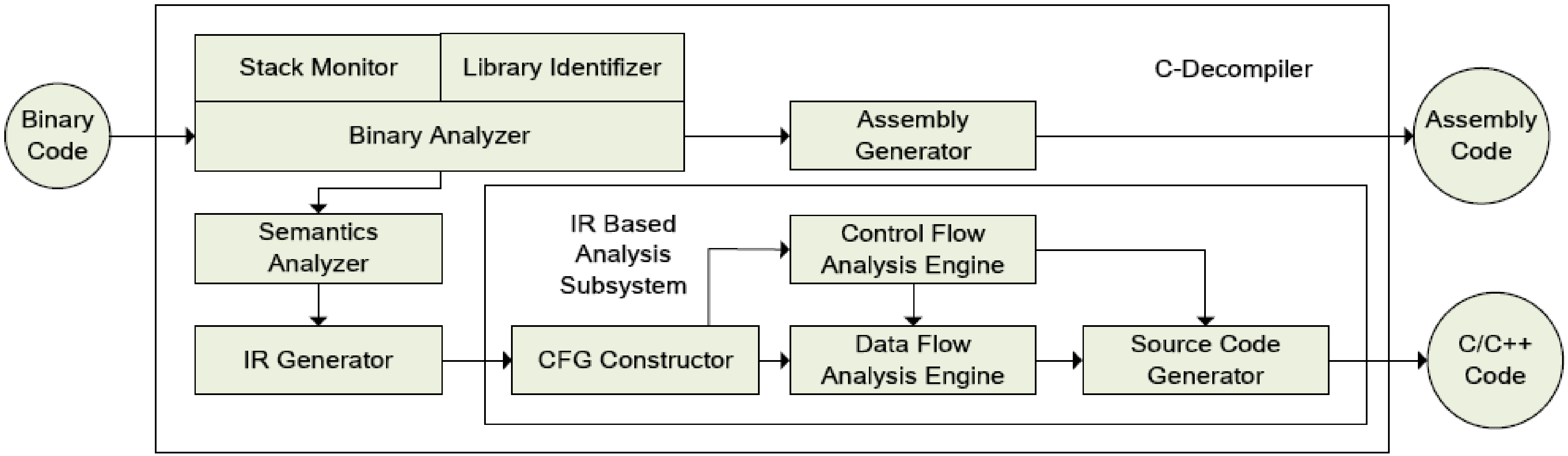
(b) Decompiled Code of IDA Hex_rays

```
int _tmain(int argc, _TCHAR argv)
{
    int loc1[2][3];
    int loc2[3][2];
    int loc3[2][2];
...

    loc1[0][0] = 1;
    loc1[0][1] = 2;
    loc1[0][2] = 3;
...

loc3[loc4][loc5] = ((loc1[loc4][loc6] *
loc2[loc6][loc5]) + loc3[loc4][loc5]);

...
```

(c) Decompiled Code of C-Decompiler

- Variables reduction
- Function Identification
- STL (C++) Identification

Architecture



Shadow Stack

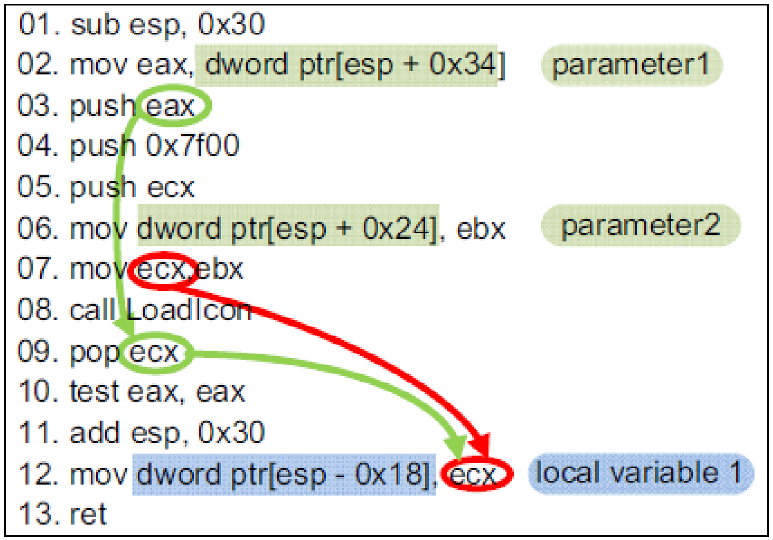


Fig. 3. An example of how the classic algorithm works. The memory locations with green mark are parameters, and blue for local variables. The red arrowed curve is the propagation path of ecx according to the classic algorithm, while the green curve is the correct path.

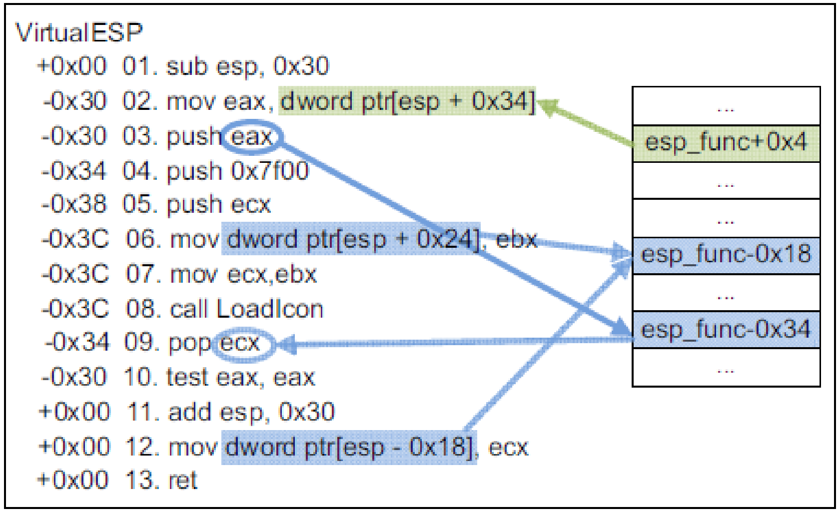


Fig. 4. With the help of the shadow stack, we can see that line 6 and line 12 write to the same memory location. Totally one parameter and two local variables are identified. Moreover, the the correct data path of ecx is recognized.

Inter-Basic-Block register propagation

Algorithm: *dcc* Register Propagation

```
procedure ExtRegCopyProp
/* Pre: dead-register analysis has been performed.
 *dead-condition code analysis has been performed.
 *register arguments have been detected.
 *function return registers have been detected.
 * Post: temporary registers are removed from the
intermediate code. */
initExpStk()
for (all basic blocks b of function ) do
 for (all instructions j in b ) do
  for (all registers r used by instruction j ) do
   if ((ud( r ) = { def }) && CanDoPropagate())
   /* uniquely defined at instruction def */
    DoPropagate();
   end if
  end for
 end for
end for
```

Algorithm: Inter-BB Register Propagation

```
procedure InterBBRegCopyProp

for (all ret instructions k in the program) do
   ConstructPath(path);
   //Construct all the instruction paths for all the ret
end for
for (all instructions j using registers r ) do
   XBB_ud( r ) = ComputeUD();
   //Compute the ud-chains based on constructed
     instruction paths
end for
for (all instructions j in function ) do
  for (all registers r used by instruction j ) do
    if ((XBB_ud( r ) = { def }) && CanDoAcrossBB())
      DoPropagate();
    end if
  end for
end for
```

```
procedure CanDoAcrossBB
if (path of r is unique)//r is only appear in one path.
  CanDoPropagate();
end if
```

Dcc vs. C-Decomplier

Binary Code	The *dcc* Decompiler	*C-Decompiler*

01 SUB eax, 2
02 JE L1
03 SUB eax, 0Dh
04 JE L2
05 SUB eax, 102h
06 JE L3
07 …
08 L1 …
09 L2 …
10 L3 …
	if ((loc0 - 2) != 0){
	if ((loc1 - 13) != 0) {
	if ((loc2 - 258) != 0) {
	…
	}else {//L3
	…}
	}else {//L2
	…}
	}else {//L1
	…}
		if ((loc0 - 2) != 0) {
		if (((loc0 - 2) - 13) != 0) {
		if ((((loc0 - 2) - 13) - 258) != 0) {
		…
		}else {//L3
		…}
		}else {//L2
		…}
		}else {//L1
		…}

Fig. 5. The comparison of the decompiled codes from the *dcc* decompiler and *C-Decompiler*. This is mainly to illuminate the difference of the common method and the inter-*BB* method. The code decompiled by the *dcc* decompiler is in the middle and the one by *C-Decompiler* is presented on the right.

STL function identification based on

(a) Original Code	(b) Assembly Code	(c) Identification
vector<int> vl;	xor esi,esi mov dword ptr [esp+18h],esi mov dword ptr [esp+1Ch],esi mov dword ptr [esp+20h],esi	vector loc1;
vl.push_back(10);	xor ecx,ecx push ecx lea eax,[esp+18h] push eax lea eax,[esp+10h] push eax lea ecx,[esp+18h] mov dword ptr[esp+3Ch],esi push ecx lea eax,[esp+24h] mov dword ptr[esp+18h],0Ah call std::vector<int,std::allocator<int> >::insert	loc1.push(10);

Fig. 8. An example of *STL* identification. Considering the original code in (a), (c) is the output of *STL* identification by *C-Decompiler* with the input of assembly code in (b).

Experiments-1

(a)Original code	(b)Decompiled code

<pre>int APIENTRY _tWinMain(...)
{
  MSG msg;
  HACCEL hAccelTable;
  LoadString(...);
  LoadString(...);
  MyRegisterClass(hInstance);
  if (!InitInstance (hInstance, nCmdShow))
  {
    return FALSE;
  }
  hAccelTable = LoadAccelerators(...));
  while (GetMessage(&msg, NULL, 0, 0))
  {
    if (!TranslateAccelerator(...))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  return (int) msg.wParam;
}</pre>
	<pre>int _tWinMain(...){
	  HACCEL loc1;
	  MSG loc2;
	  int loc3; /* eax */
	  LoadStringW (...);
	  LoadStringW (...);
	  proc_1 (hInstance);
	  if (proc_2 (hInstance, nCmdShow) == 0) {
	    loc3 = 0;
	  }
	  else {
	    loc1 = LoadAcceleratorsW (...);
	    while((GetMessageW (loc2,...) != 0)) {
	      if(TranslateAcceleratorW (...)== 0){
	        TranslateMessage (&loc2);
	        DispatchMessageW (&loc2);
	      }
	    } /* end of while */
	    loc3 = loc2.wParam;
	  }
	  return (loc3);
	}</pre>

Experiment-2

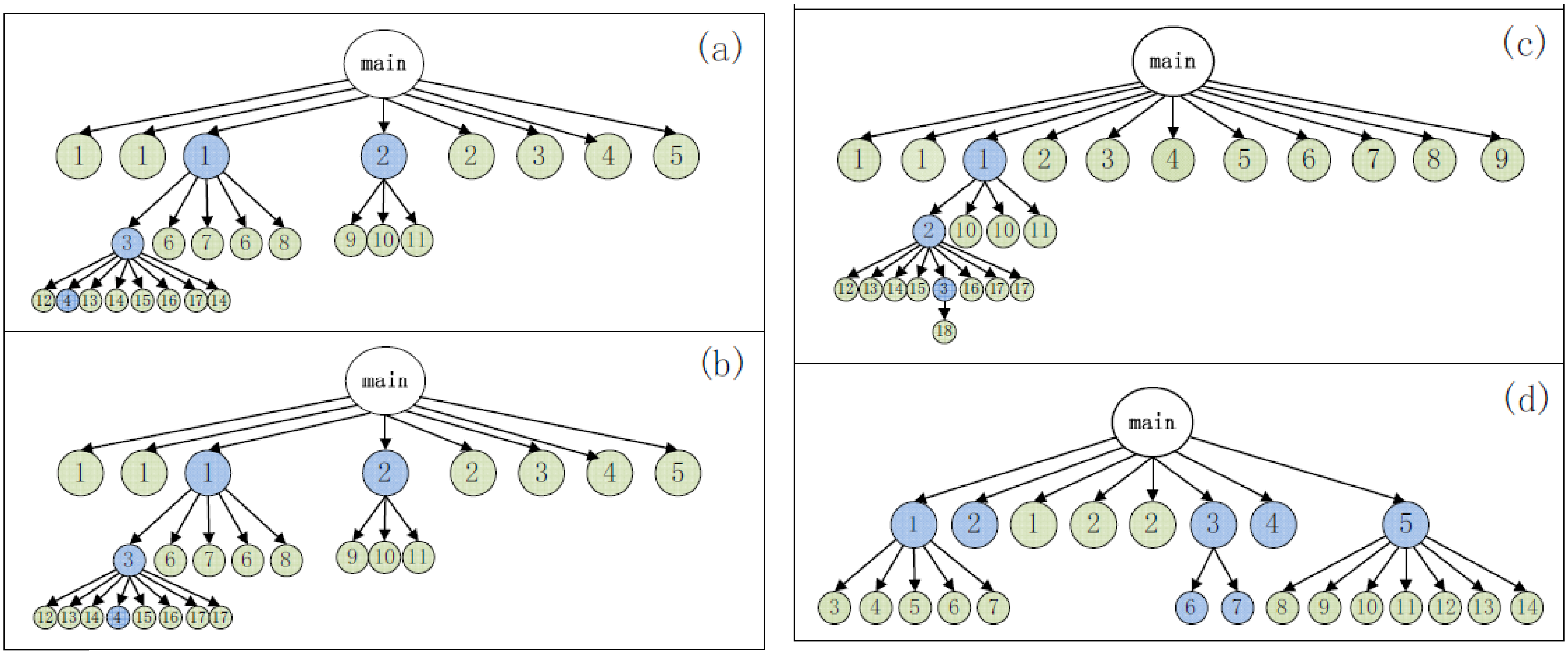


Fig. 10. Function-call trees of the code decompiled. (a), (b), (c) and (d) are the function-call trees of the original code, C-Decompiler, Hex_rays and Boomerang respectively. The nodes are functions. The green nodes represent APIs, and the blue ones stand for *UDFs*.

Reduction Rate

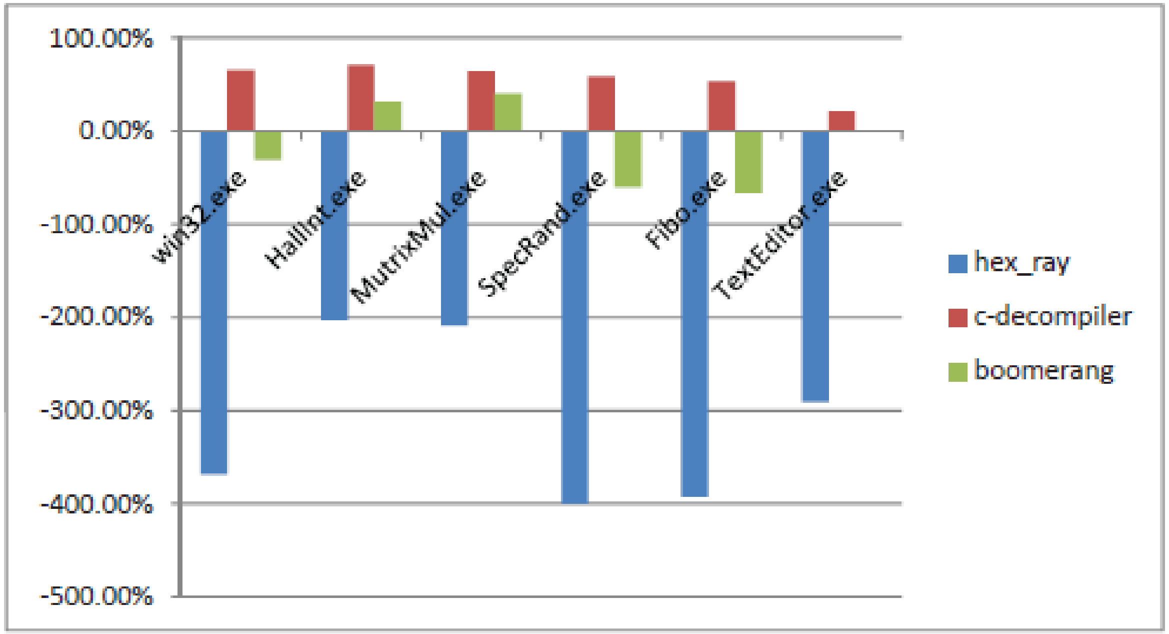


Fig. 14. Summary of reduction rate of the 3 decompilers. The red, green and blue bars stand for the *reduction%* of *C-Decompiler*, Boomerang and Hex_rays respectively. The higher bars mean the better performance. Generally speaking, the red bar is the highest, which means the length of the code decompiled by C-Decompiler is closest to the length of the original code.

Variable expansion rate

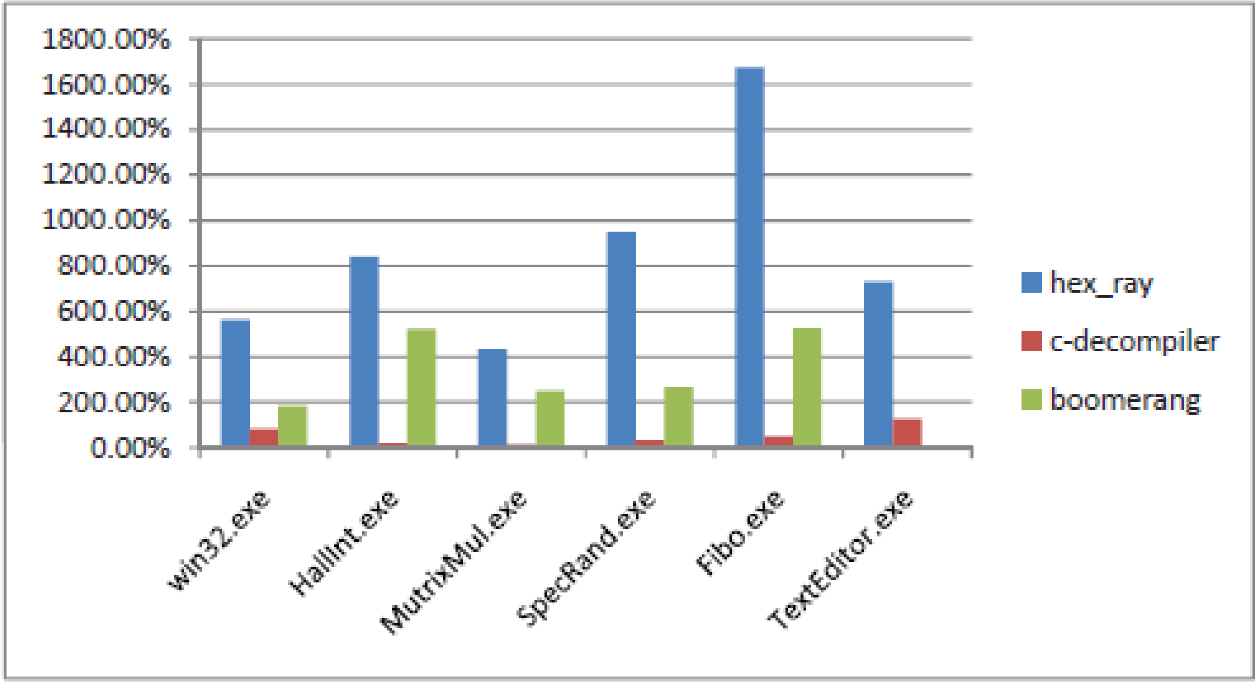


Fig. 15. Summary of variable expansion rate. The relationship of the colors and the decompilers is the same to the Figure 15. The lower bars present the better performance. Generally speaking, the red bar is the lowest. This means the quantity of variables in the code decompiled by C-Decompiler is the closest to the one of the original code.

Agenda

- Introduction
- An Online Model Checker for Distributed Systems [with Microsoft Research Asia]
- A Refined Decompiler to Generate C/C++ Code with High Readability [WCRE 2010]
- Some current research topics in our group [call for cooperation]

1. Mixing Lockset Analysis and Symbolic Execution for Critical Section Inference [submitted to ACM APSYS 2011]

- Problem: How to guarantee the lock-unlock pair?
- Idea: Use KLEE [OSDI 2008] to reason the lock in large system.

```
if (x)
  lock(L)
...
if (x)
  unlock(L)
```
(a) The simplest case

```
if (x)
  lock(L)
y = x
if (y)
  unlock(L)
```
(b) Flow-sensitive

```
if (x)                foo(int y) {
  lock(L)               if (y)
...                       unlock(L)
foo(x)                }
```
(c) Context-sensitive

Figure 1: Variations of a common pattern seen in Apache (simplified for the purpose of illustration)

Solution: combine scalable lockset analysis, which identifies functions with ambiguous locksets, and accurate symbolic execution, which resolves the ambiguity of these functions locally, for better analysis results.

Primary result

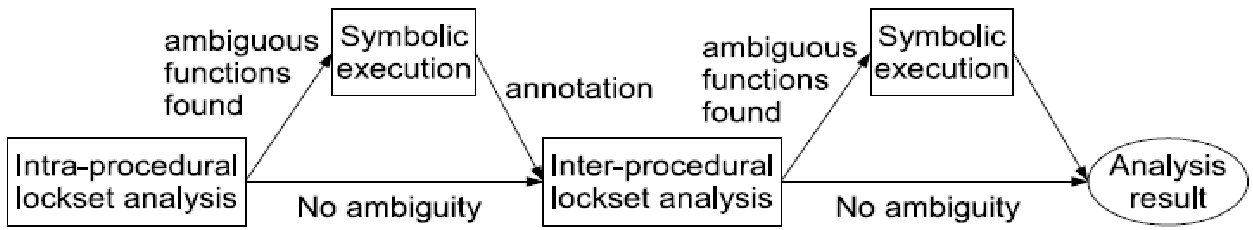


Figure 2: The analysis procedure

```
safe_mutex_lock(fifo->mut);
while (fifo->full) {
  ...
  if (syncGetTerminateFlag() != 0) {
    pthread_mutex_unlock(fifo->mut);
    close(hInfile);
    return -1;
  }
}
...
if (queueElement == NULL) {
  close(hInfile);
  handle_error(EF_EXIT,-1,"pbzip2:...");
  return -1;
}
...
safe_mutex_unlock(fifo->mut);
```

Not match with fifo-mut

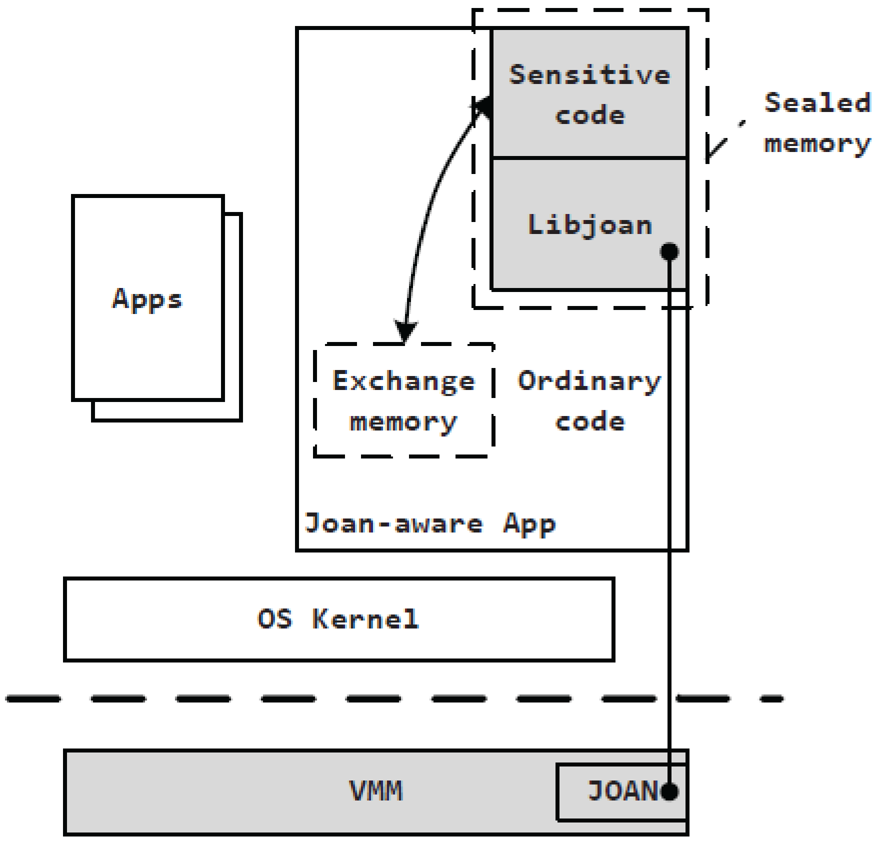
Figure 4: Mismatch of lock/unlock in Pbzip2

name	instruction	paths	time	result
ap_buffered_log_writer*	10,609	6	2.9s	yes
cgi_bucket_read*	-	-	-	-
child_main*	-	-	-	-
apr_file_seek	12,388	33	6.3s	yes
apr_file_read*	9,976	3	2.2s	yes
apr_file_flush	10,195	7	4.3s	yes
apr_file_write*	-	-	-	-
apr_file_gets*	10,466	7	1.3s	yes
proc_mutex_proc_pthread_create	-	-	-	-
proc_mutex_proc_pthread_cleanup	10,152	5	2.3s	no
allocator_free*	9,877	2	3.3s	yes
allocator_alloc	10,861	21	160s	yes
apr_pool_create_ex*	-	-	-	-
apr_pool_destroy*	-	-	-	-
apr_pollset_poll*	10,010	3	2.8s	yes
apr_pollset_add	10,102	3	2.3s	yes

Table 1: Symbolic execution of Apache functions with ambiguous locksets

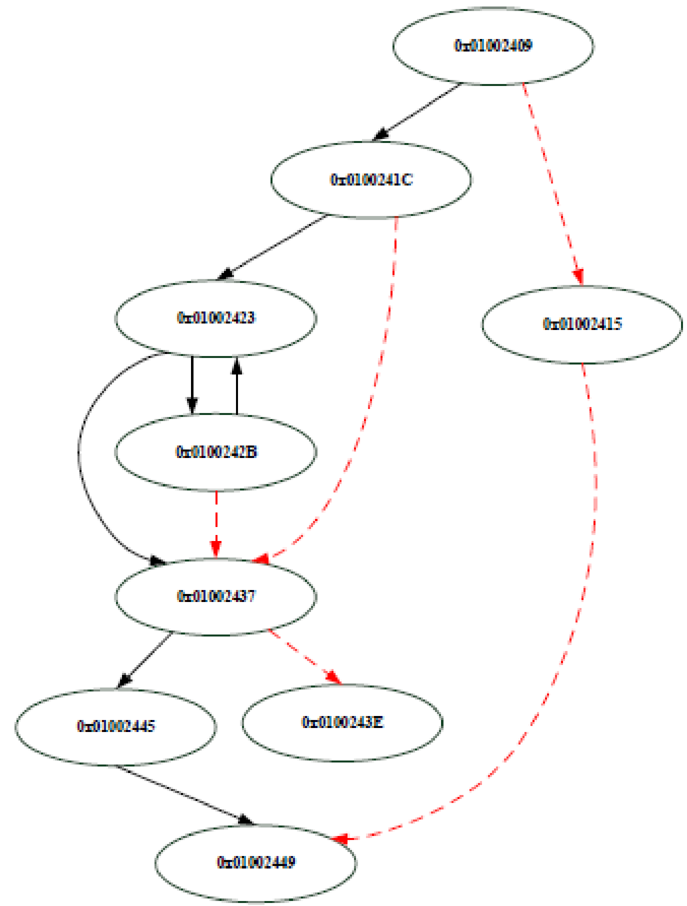
2. Shepherd application privacy with virtualized special purpose memory [OSDI 2010 Poster]

- Reduce the TCB to include only user-selected sensitive code and the hypervisor assisted by taint analys
- Exploit memory virtualization to provid privacy aware memory primitives.



3. Complete CFG by static analysis

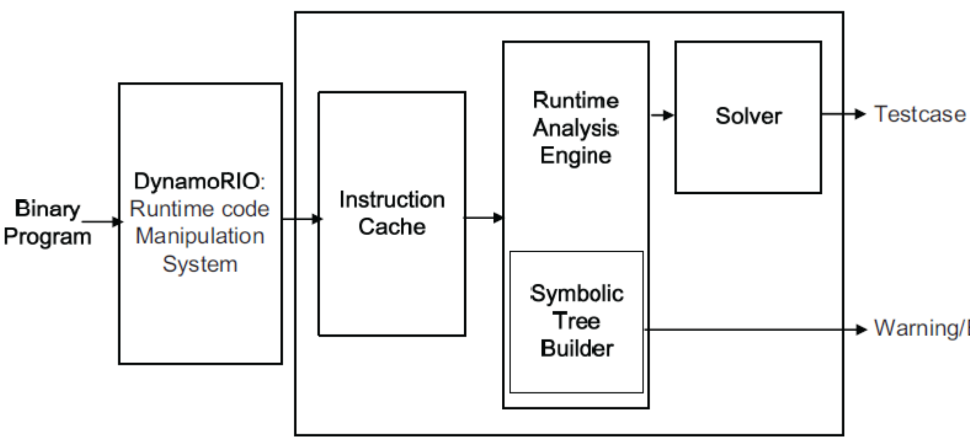
- *Dynamic Taint Analysis to get a real execution path*
- *Static analysis to complete the execution path as a CFG.*
- *In order to reason the key path in a large system*



The concrete black path indicates the real execution path, and the dotted red ones are supplemented by the static analyzer.

4. Binary Symbolic Execution tool

- A dynamic symbolic execution tool, for x86 binary code. It’s based on the DynamoRIO as a frontend
- Combine program slicing and dynamic taint analysis
- A tool to reason the real binary code



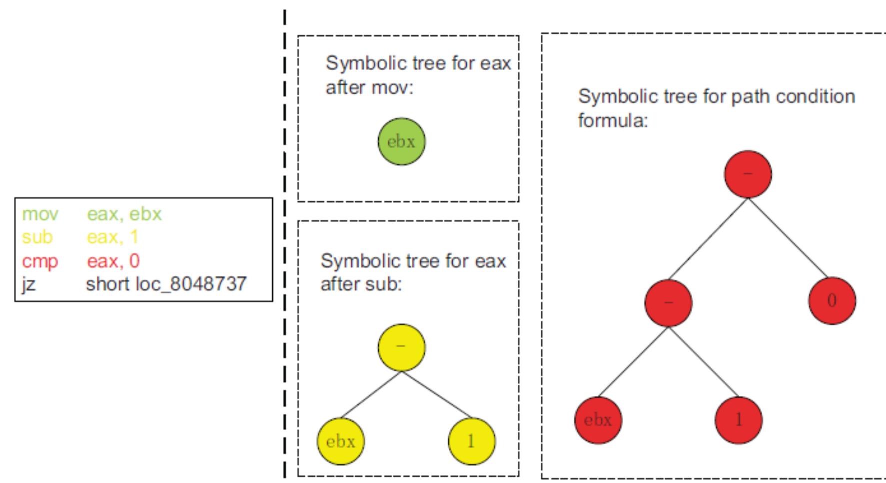
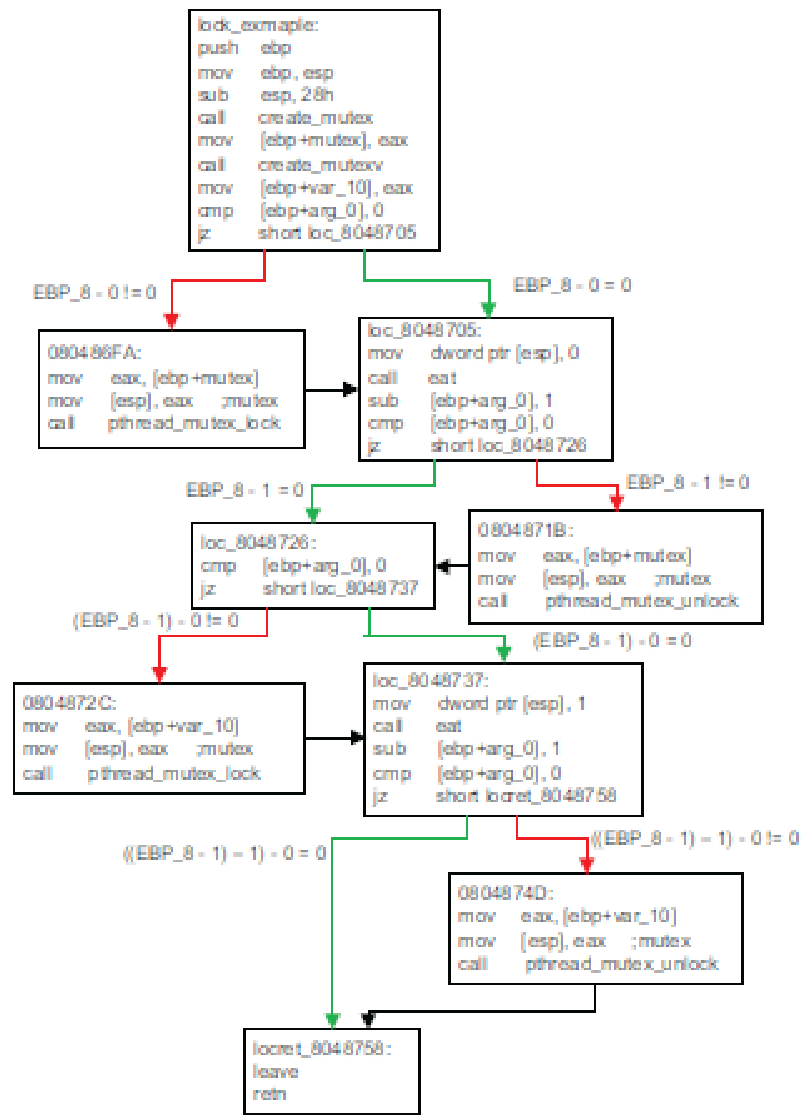


Figure 2: Example of Symbolic Tree.

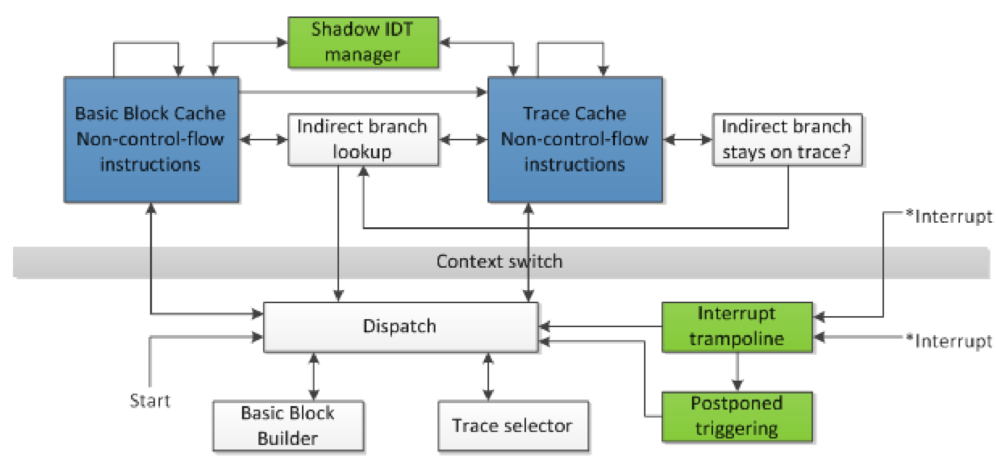
Case study

```
void lock_example(int multi_thread)  {
        pthread_mutex_t *forkA = create_mutex();
        pthread_mutex_t *forkB = create_mutex();
        if(multi_thread) pthread_mutex_lock(forkA);
        eat(0);
        multi_thread--;
        if(multi_thread) pthread_mutex_unlock(forkA);
        if(multi_thread) pthread_mutex_lock(forkB);
        eat(1);
        multi_thread--;
        if(multi_thread) pthread_mutex_unlock(forkB);
        return;
}
```



5. System wide real code analysis

- Bitblaze[ICISS 08] use a simulator QEMU to do system-wide analysis.
- A real hardware supported analysis may be more practical
  - DynamoRIO runs in user-level
  - how about in a supervisor-level tool?
  - just a proposal



Reference

- Zhengwei Qi, Liang Liu, Alei Liang, Hao Wang, Ying Chen: An Online Model Checking Tool for Safety and Liveness Bugs. ICPADS 2008: 493-500
- Rob Gerth, Doron Peled, Moshe Y. Vardi, Pierre Wolper: Simple on-the-fly automatic verification of linear temporal logic. PSTV 1995: 3-18
- Gengbiao Chen, Zhuo Wang, Ruoyu Zhang, Kan Zhou, Shiqiu Huang, Kangqi Ni, Zhengwei Qi, Kai Chen, Haibing Guan: A Refined Decompiler to Generate C Code with High Readability. WCRE 2010: 150-154
- Cristian Cadar, Daniel Dunbar, Dawson R. Engler: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI 2008: 209-224
- Derek Bruening. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D Thesis, MIT, September 2004
- Dawn Song, David Brumley, Heng Yin, et al. BitBlaze: A New Approach to Computer Security via Binary Analysis. ISISS 2008

Questions?

THANK YOU !

http:// 202.120.40.124