



# Intel® Inspector XE 2016

**Memory and thread debugger**



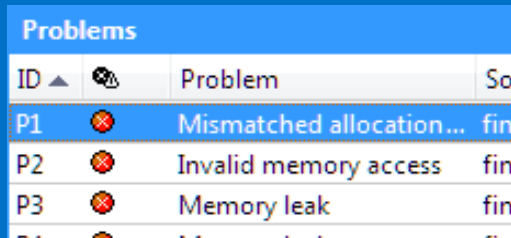
# Agenda

- Intro to Intel® Inspector XE
- Analysis workflow
- Memory problem analysis
- Threading problem Analysis
- Integration with debugger
- Automated regression testing and user API

# Intro to Intel® Inspector XE

# Motivation for The Inspector XE

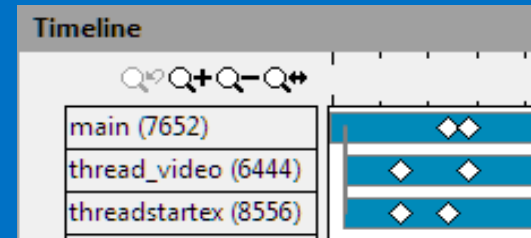
## Memory Errors



ID	Problem	Source
P1	Mismatched allocation...	fin
P2	Invalid memory access	fin
P3	Memory leak	fin

- Invalid Accesses
- Memory Leaks
- Uninitialized Memory Accesses

## Threading Errors



- Data Races
- Deadlocks
- Cross Stack References

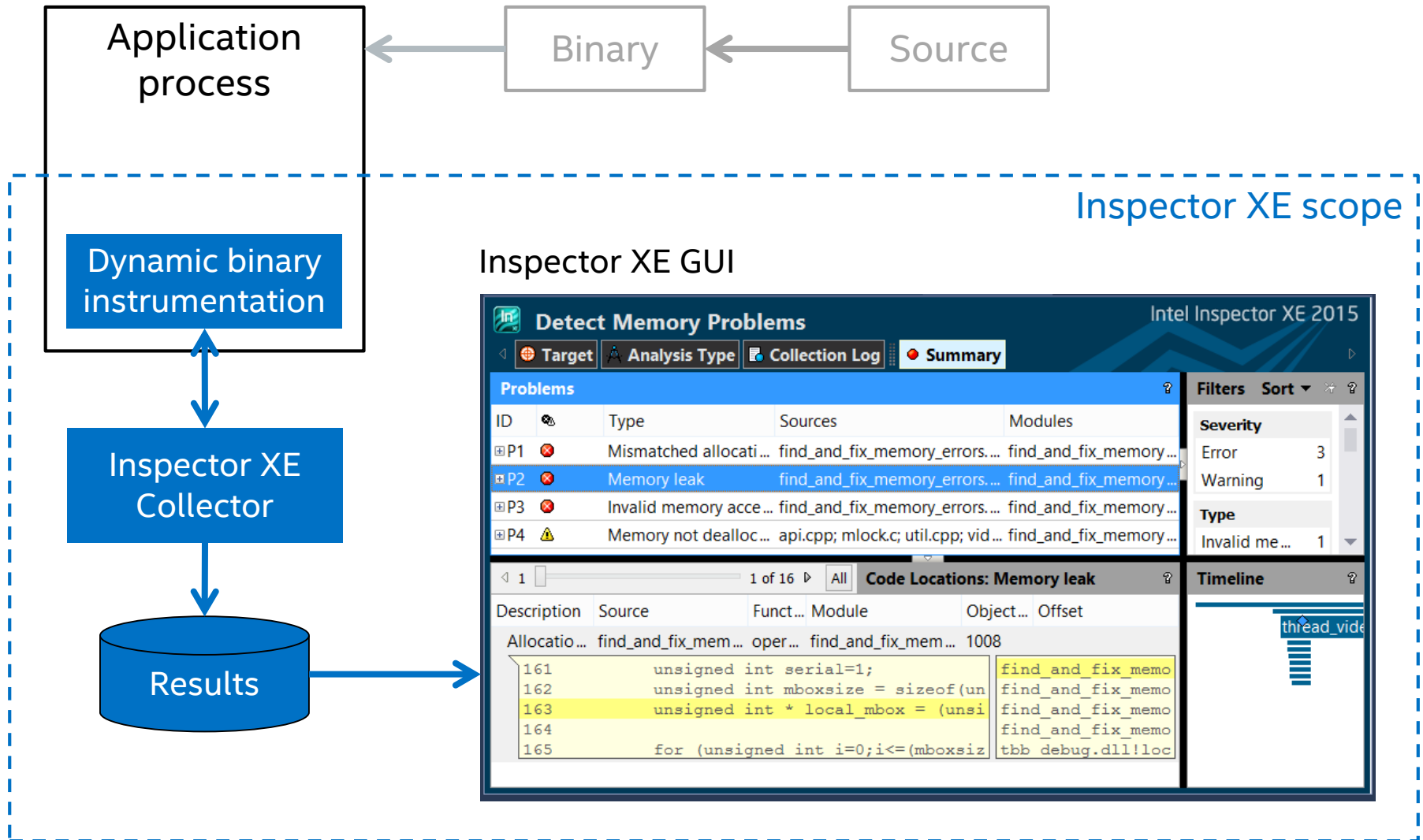
## Multi-threading problems

- Hard to reproduce,
- Difficult to debug
- Expensive to fix



Let the tool do it for you

# Intel Inspector XE: Dynamic analysis



# How it looks: Visual Studio\* Integration

**Run analysis from toolbar**

**Problems found: memory leaks**

**Choose existing project, no special configuration**

**Memory allocation site in source code**

**Call stack**

ID	Type	Modules
P1	Mismatched all...	find_and_fix_memory...
P2	Memory leak	find_and_fix_memory...
P3	Invalid memory acce...	find_and_fix_memory...
P4	Memory not dealloc...	api.cpp; mlock.c; util.cpp; vid... find_and_fix_memory...

```
161 unsigned int serial=1;
162 unsigned int mboxsize = sizeof(un
163 unsigned int * local_mbox = (unsi
164
165 for (unsigned int i=0;i<=(mboxsiz
```

thread\_vide

# Standalone GUI: Windows\* and Linux\*

The screenshot shows the Intel Inspector XE 2015 standalone GUI. The window title is "Intel Inspector XE 2015". The main menu includes "File", "View", and "Help". The "Project Navigator" on the left shows a project named "My Inspector XE" with sub-items "r000ti2", "r001ti2", "r002ti2", and "r003ti2". The main area is titled "Configure Analysis Type" and features a "Memory Error Analysis" dropdown menu. A slider for "Analysis Time Overhead" is set to "10x-40x", and a "Memory Overhead" bar chart is visible. The "Detect Memory Problems" section includes a "Copy" button and several checked options: "Detect memory leaks upon application exit", "Detect resource leaks", and "Enable interactive memory growth detection". On the right, a "Start" button is highlighted, along with "Stop", "Close", "Reset Growth Tracking", "Measure Growth", "Reset Leak Tracking", and "Find Leaks".

Open existing project or create a new one

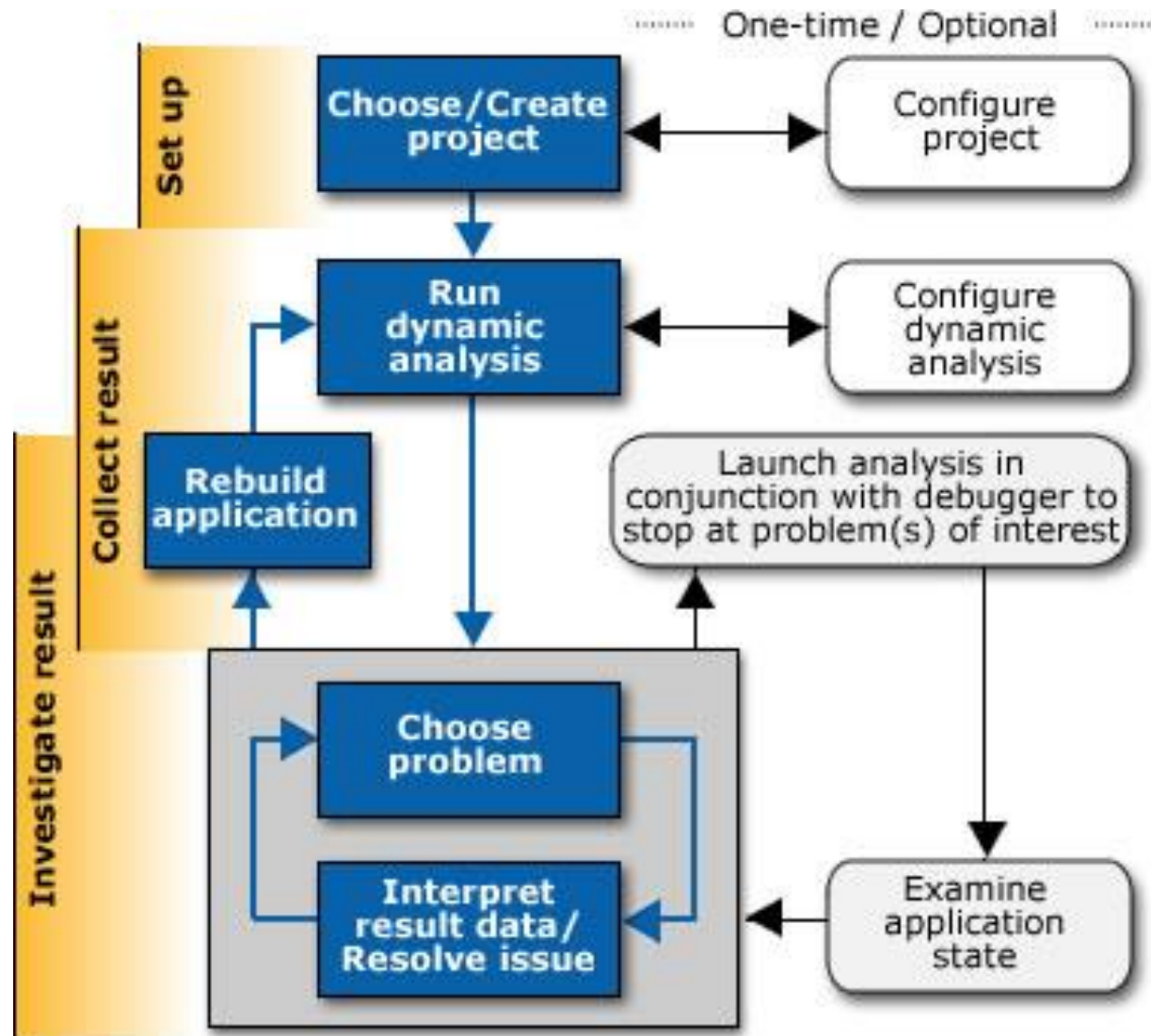
Choose analysis type

Start analysis

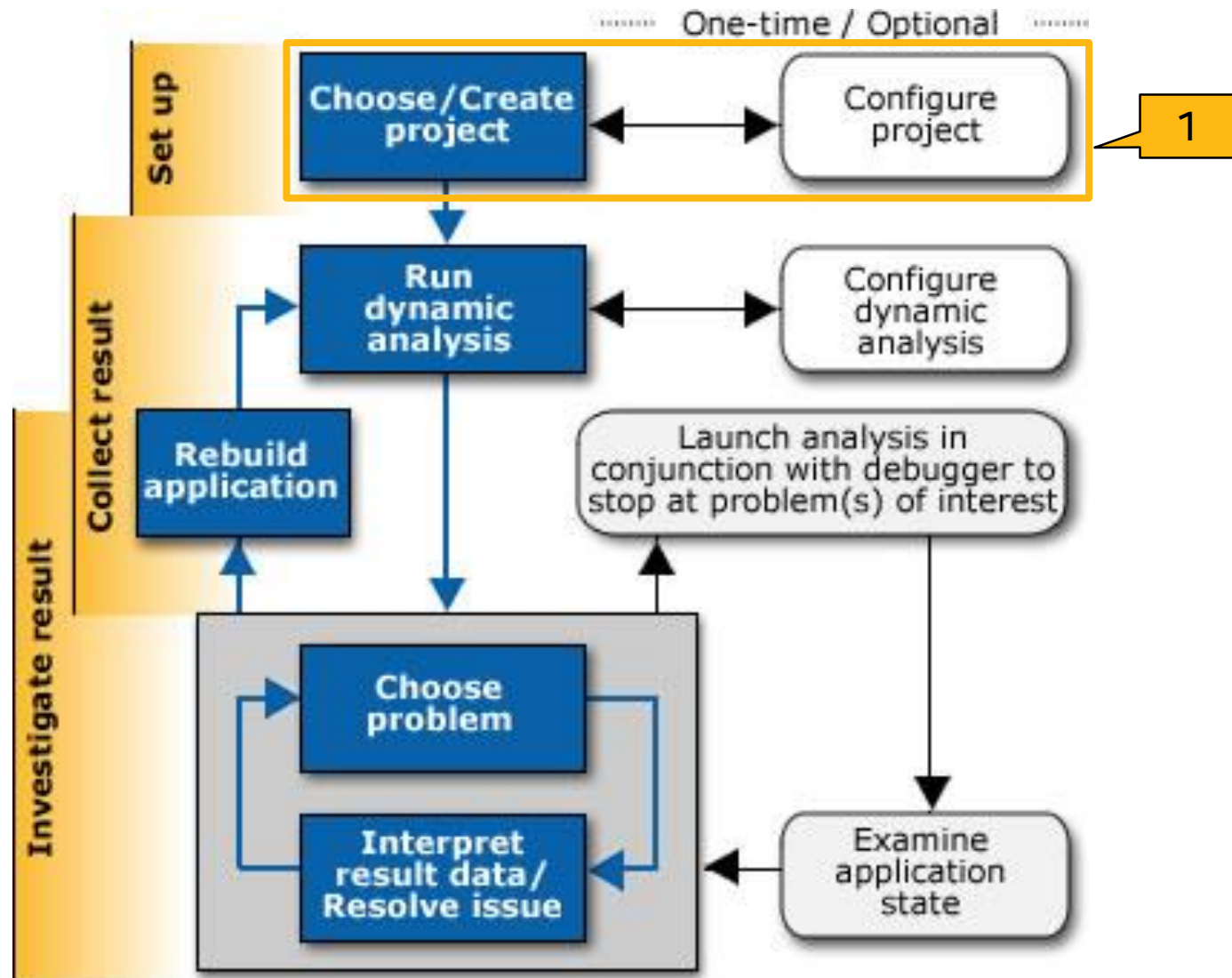
# Analysis workflow



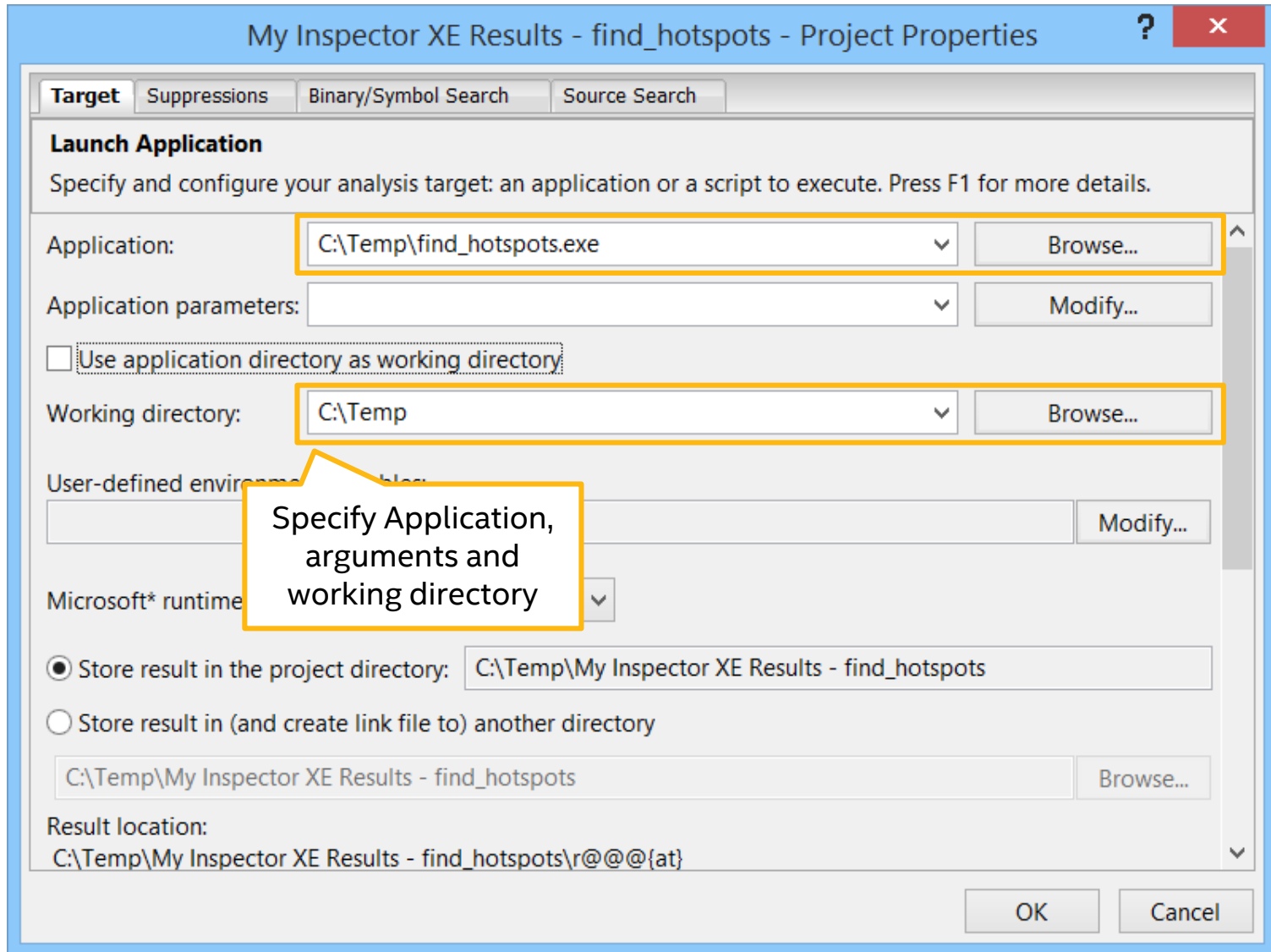
# Workflow: Dynamic Analysis



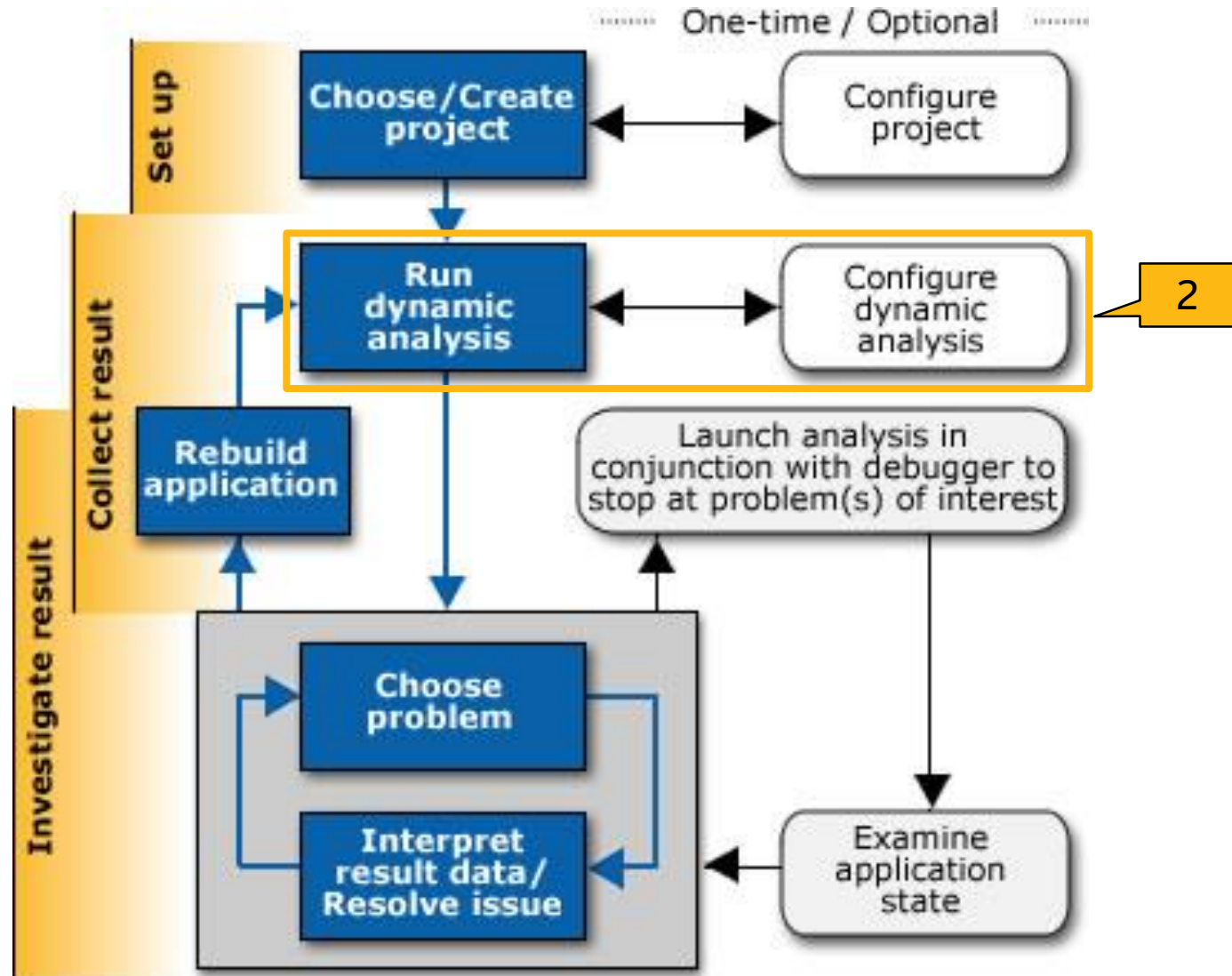
# Workflow: Dynamic Analysis



# Workflow: setup project



# Workflow: Dynamic Analysis



# Workflow: select analysis and start

**Configure Analysis Type** Intel Inspector XE 2015

**Analysis Type**

- Threading Error Analysis
- Memory Error Analysis
- Threading Error Analysis
- Custom Analysis Types

Analysis Time Overhead	Memory Overhead
10x-40x	Detect Deadlocks
20x-80x	Detect Deadlocks and Data Races
40x-160x	Locate Deadlocks and Data Races

**Locate Deadlocks and Data Races** Copy

Widest scope threading error analysis type. Maximizes the load on the system and the time and resources required to perform analysis; however, detects the widest set of errors and provides context and maximum detail for those errors. Press F1 for more details.

Terminate on deadlock

Stack frame depth: 16

Scope: Normal

Remove duplicates

Use maximum resources

**Start**

Stop

Close

Reset Growth Tracking

Measure Growth

Reset Leak Tracking

Find Leaks

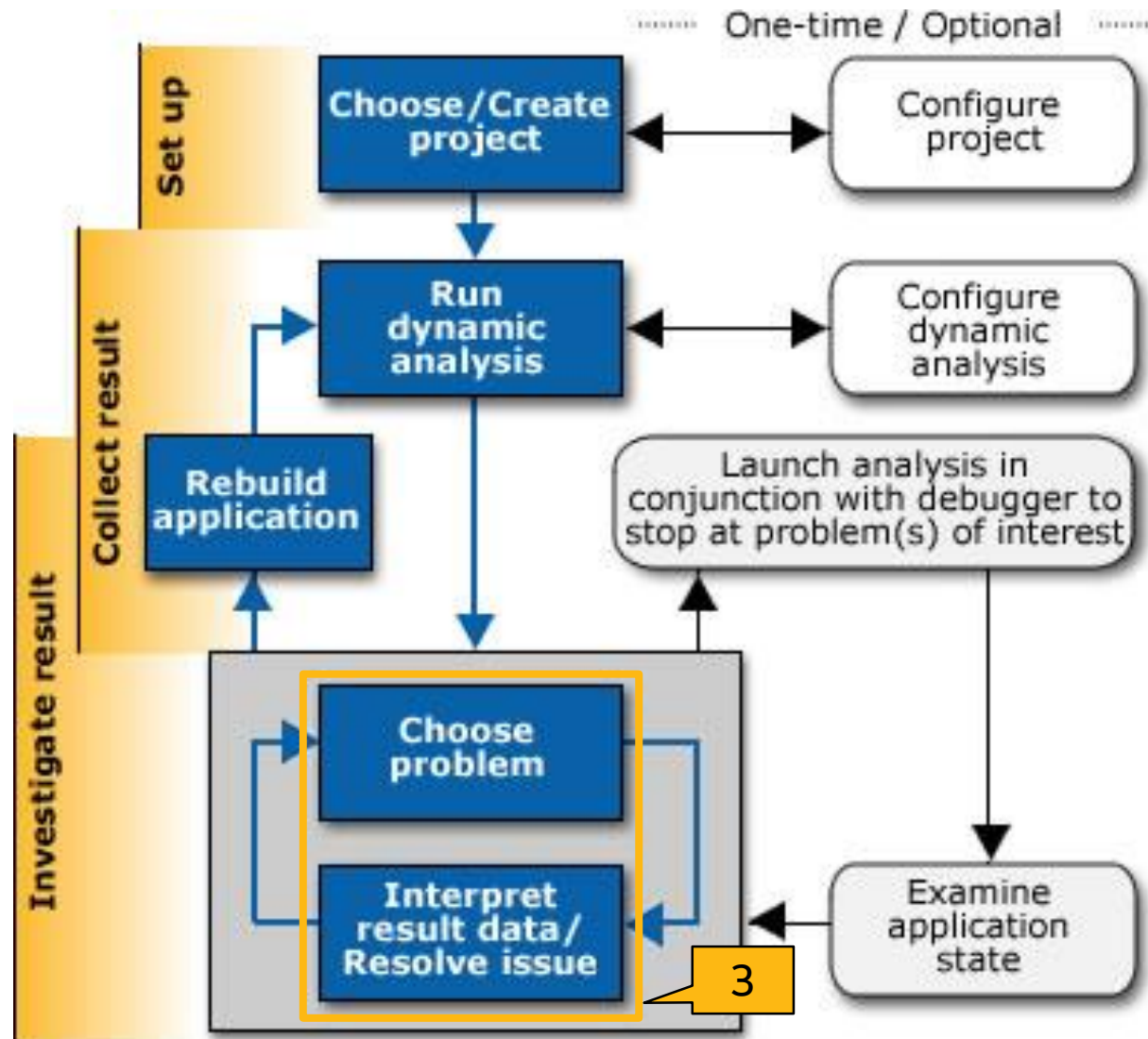
Project Properties...

Command Line...

**1. Select Analysis Type**

**2. Click Start**

# Workflow: Dynamic Analysis





# Workflow: manage results

**Intel Inspector XE 2015**

**Detect Deadlocks and Data Races**

Target | Analysis Type | Collection Log | Summary

**Problems**

ID	Type	File	Module	Status
P1	Data race	find_and_fix_threading_errors.cp...	find_and_fix_threading_errors.exe	New
P2	Data race	winvideo.h	find_and_fix_threading_errors.exe	New
	Data race	winvideo.h:270	find_and_fix_threading_errors.exe	New
	Data race	winvideo.h:270	find_and_fix_threading_errors.exe	New
	Data race	winvideo.h:201; winvideo.h:270	find_and_fix_threading_errors.exe	New

**Code Locations: Data race**

Read winvideo.h:270 next\_frame find\_and\_fix\_threading\_errors.exe

```
268 {
269     if(!running) return false;
270     g_updates++; // Fast but inaccura
271     if(!threaded) while(loop_once(thi
272     else if(g handles[1]) {
```

**Timeline**

- main (4960)
- thread\_video (4672)
- TBB Worker Thread (2848)
- TBB Worker Thread (1724)
- TBB Worker Thread (6004)

Read: winvideo.h:270  
Write: winvideo.h:270

**Annotations:**

- Double click on Problem to navigate to source
- Powerful filtration feature
- Code locations grouped into Problems to simplify results management

# Workflow: navigate to sources



# Memory problem analysis

# Memory problem Analysis

## Analyzed as software runs

- Data (workload) -driven execution
- Program can be single or multi-threaded
- Diagnostics reported incrementally as they occur

## Includes monitoring of:

- Memory allocation and allocating functions
- Memory deallocation and deallocating functions
- Memory leak reporting
- Inconsistent memory API usage

## Analysis scope

- Native code only: C, C++, Fortran
- Code path must be executed to be analyzed
- Workload size affects ability to detect a problem

# Memory problems

## Memory leak

- a block of memory is allocated
- never deallocated
- not reachable (there is no pointer available to deallocate the block)
- Severity level = **(Error)**

```
// Memory leak
```

```
char *pStr = (char*) malloc(512);  
return;
```

## Memory not deallocated

- a block of memory is allocated
- never deallocated
- still reachable at application exit (there is a pointer available to deallocate the block).
- Severity level = **(Warning)**

```
// Memory not deallocated
```

```
static char *pStr = malloc(512);  
return;
```

## Memory growth

- a block of memory is allocated
- not deallocated, within a specific time segment during application execution.
- Severity level = **(Warning)**

```
// Memory growth
```

```
// Start measuring growth  
static char *pStr = malloc(512);  
// Stop measuring growth
```

# Memory problems

## Uninitialized memory access

- Read of an uninitialized memory location

```
// Uninitialized Memory Access

void func()
{
    int a;
    int b = a * 4;
}
```

## Invalid Memory Access

- Read or write instruction references memory that is logically or physically invalid

```
// Invalid Memory Access

char *pStr = (char*) malloc(20);
free(pStr);
strcpy(pStr, "my string");
```

## Kernel Resource Leak

- Kernel object handle is created but never closed

```
// Kernel Resource Leak

HANDLE hThread = CreateThread(0,
    8192, work0, NULL, 0, NULL);
return;
```

## GDI Resource Leak

- GDI object is created but never deleted

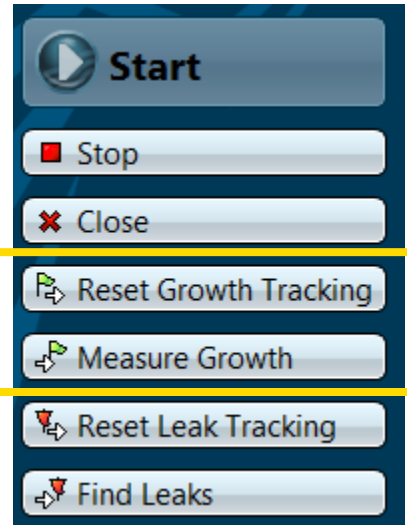
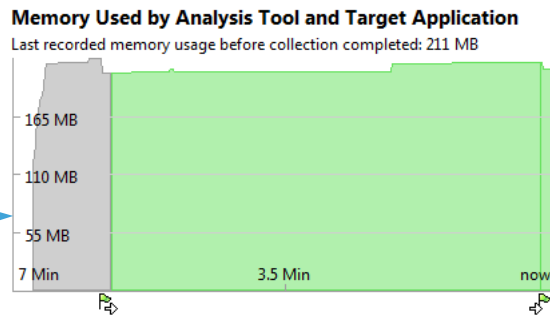
```
// GDI Resource Leak

HPEN pen = CreatePen(0, 0, 0);
return;
```

# Incrementally Diagnose Memory Growth

As your app is running...

Memory usage graph plots memory growth



Select a cause of memory growth

ID	Type	Sources	Modules	Object Size	State
	Memory growth	gdiplus.dll:0x47240	gdiplus.dll	40960	New
	Memory growth	find_and_fix_memory_errors.cpp:163	find_and_fix_memory_errors.exe	90108	Not fixed
	Memory growth	find_and_fix_memory_errors.cpp:163	find_and_fix_memory_errors.exe	1802160	Not fixed
	Memory growth	find_and_fix_memory_errors.cpp:163	find_and_fix_memory_errors.exe	30036	Not fixed
	Memory growth	find_and_fix_memory_errors.cpp:163	find_and_fix_memory_errors.exe	1621944	Not fixed
	Memory growth	find_and_fix_memory_errors.cpp:170	find_and_fix_memory_errors.exe	40	Not fixed

Description	Source	Function	Module	Object Size	Offset
Allocation site	find_and_fix_memory_errors.cpp:163	operator()	find_and_fix_memory_errors.exe	90108	
	161 unsigned int serial=1;		find_and_fix_memory_errors.exe		
	162 unsigned int mboxsize = sizeof(unsigned int)*(max_objectid() +		find_and_fix_memory_errors.exe		
	163 unsigned int * local_mbox = (unsigned int *) malloc(mboxsize);		find_and_fix_memory_errors.exe		
	164		find_and_fix_memory_errors.exe		
	165 for (unsigned int i=0;i<=(mboxsize/(sizeof(unsigned int)));i++		tbb_debug.dll!local_wait_for_a		

See the code snippet & call stack

# Threading problem analysis

# Threading problem Analysis

## Analyzed as software runs

- Data (workload) -driven execution
- Program needs to be multi-threaded
- Diagnostics reported incrementally as they occur

## Includes monitoring of:

- Thread and Sync APIs used
- Thread execution order
  - Scheduler impacts results
- Memory accesses between threads

## Analysis scope

- Native code: C, C++, Fortran
- Managed or mixed code: C# (.NET 2.0 to 3.5, .NET 4.0 with limitations)
- Code path must be executed to be analyzed
- Workload size doesn't affect ability to detect a problem

# Data race

```
CRITICAL_SECTION cs;           // Preparation
int *p = malloc(sizeof(int)); // Allocation Site
*p = 0;
InitializeCriticalSection(&cs);
```

## Write -> Write Data Race

Thread #1

```
*p = 1; // First Write
```

Thread #2

```
EnterCriticalSection(&cs);
*p = 2; // Second Write
LeaveCriticalSection(&cs);
```

## Read -> Write Data Race

Thread #1

```
int x;
x = *p; // Read
```

Thread #2

```
EnterCriticalSection(&cs);
*p = 2; // Write
LeaveCriticalSection(&cs);
```



# Deadlock

```
CRITICAL_SECTION cs1;  
CRITICAL_SECTION cs2;  
int x = 0;  
int y = 0;  
InitializeCriticalSection(&cs1); // Allocation Site (cs1)  
InitializeCriticalSection(&cs2); // Allocation Site (cs2)
```

## Thread #1

```
EnterCriticalSection(&cs1);  
x++;  
    EnterCriticalSection(&cs2);  
    y++;  
    LeaveCriticalSection(&cs2);  
LeaveCriticalSection(&cs1);
```

## Thread #2

```
EnterCriticalSection(&cs2);  
y++;  
    EnterCriticalSection(&cs1);  
    x++;  
    LeaveCriticalSection(&cs1);  
LeaveCriticalSection(&cs2);
```

## Deadlock

1. EnterCriticalSection(&cs1); in thread #1
2. EnterCriticalSection(&cs2); in thread #2

## Lock Hierarchy Violation

1. EnterCriticalSection(&cs1); in thread #1
2. EnterCriticalSection(&cs2); in thread #1
3. EnterCriticalSection(&cs2); in thread #2
4. EnterCriticalSection(&cs1); in thread #2

# Cross-thread Stack Access

```
// A pointer visible for two threads
int *p;
CreateThread(..., thread #1, ...);
CreateThread(..., thread #2, ...);
```

Thread #1

```
// Allocated on Thread #1's stack
int q[1024];
p = q;
q[0] = 1;
```

Thread #2

```
// Thread #1's stack accessed
*p = 2;
```

# Integration with debugger

# Debugger integration

## Break into debugger

- Analysis can stop when it detects a problem
- User is put into a standard debugging session

## Windows\*

- Microsoft\* Visual Studio Debugger (vs2012 – vs2015)

## Linux\*

- gdb

2x-20x Detect Leaks  
10x-40x Detect Memory Problems  
20x-80x Locate Memory Problems

Analysis Time Overhead

**Detect Memory Problems** Copy

Medium scope memory error analysis type. Increases the load on the system and the time and resources required to perform analysis. Press F1 for more details.

- Analyze without debugger  
Run an analysis and report all detected problems. Use to view correctness issues without stopping in the debugger to examine them.
- Enable debugger when problem detected  
Run an analysis under the debugger and stop every time a problem is detected. Use to allow investigation of every problem detected.
- Select analysis start location with debugger  
Run target application under the debugger with analysis disabled until you choose to turn on analysis. Before starting, set a code breakpoint to stop execution prior to where you want analysis to begin. Sele...

# Debug this problem

Intel Inspector XE 2015

Target Analysis Type Collection Log Summary

**Problems**

ID	Type	State
P1	Memory leak	New
P2	Invalid memory access	Not f

Right click on a problem

**Filters** Sort

Severity

Error	2
memory access	1
leak	1

View Source  
Edit Source  
Copy to Clipboard  
Explain Problem  
Create Problem Report...  
**Debug This Problem**  
Change State  
Merge S

Inspector XE will set breakpoint, and launch debug session at the place of the problem occurrence

Code Local

Description	Source	Function	Module	Object
Write	mc.cpp:150	main	mc.exe	

```
148  
149     for (unsigned int i = 0;  
150         local_mbox[i] = 0;  
151  
152     return 0;
```

# Debug this problem

The screenshot displays the Intel Inspector XE interface. The main window shows the source code for `mc.cpp` in the `main()` function. The code is as follows:

```
{
    unsigned int max_objectid = 28;
    unsigned int mboxsize = sizeof(unsigned int)*max_objectid;
    unsigned int * local_mbox = (unsigned int *)malloc(mboxsize);

    for (unsigned int i = 0; i <= (mboxsize / (sizeof(unsigned int))); i++)
        local_mbox[i] = 0;
```

A yellow callout points to the line `local_mbox[i] = 0;` with the text: "Problematic code location with context values". Below this line, a tooltip shows the value of `local_mbox[i]` as `4261281277`.

The Autos window shows the following local variable values:

Name	Value	Type
i	28	unsigned int
local_mbox	0x002e5a50 {0}	unsigned int *
local_mbox[i]	4261281277	unsigned int
mboxsize	112	unsigned int

A yellow callout points to the Autos window with the text: "Local variable values".

The Problem Details window shows the error message: "Invalid memory access at 0x002e5ac0 for thread 5088". A yellow callout points to this message with the text: "Inspector XE problem context". Below the error message, the source code context is shown:

```
148
149     for (unsigned int i = 0; i <= (mbox
150         local_mbox[i] = 0;
151
```

# Debugger options

Memory Error Analysis

2x-20x 10x-40x 20x-80x

Detect Leaks

Detect Memory Problems

Locate Memory Problems

Analysis Time Overhead

Memory Overhead

**Detect Memory Problems** Copy

Medium scope memory error analysis type. Increases the load on the system and the time and resources required to perform analysis. Press F1 for more details.

- Analyze without debugger  
Run an analysis and report all detected correctness issues without stopping the target.
- Enable debugger when problem detected  
Run an analysis under the debugger and stop every time a problem is detected. Use to allow investigation of every problem detected.
- Select analysis start location with debugger  
Run target application under the debugger with analysis disabled until you choose to turn on analysis. Before starting, set a code breakpoint to stop execution prior to the start of the analysis.

Start debugger session for each problem detected

Inspector XE starts analysis only after passing a breakpoint

# Regression testing and user API



# Automate Regression Analysis

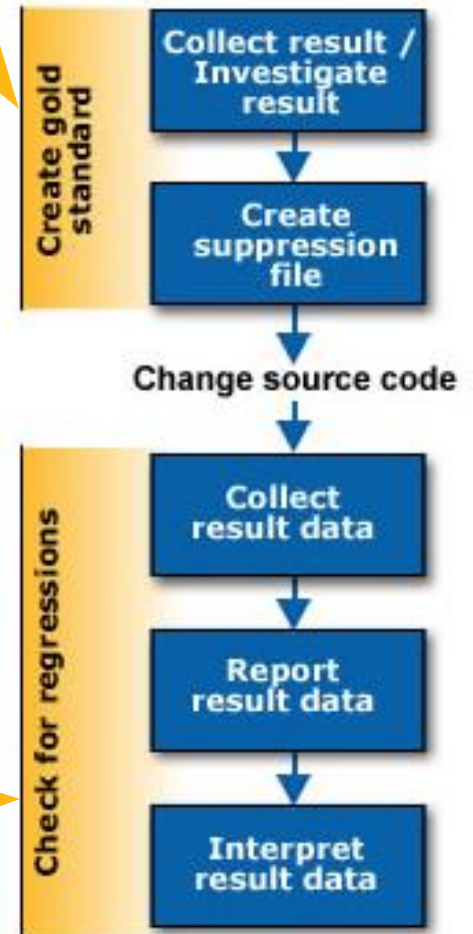
## Data collection from script

- Command line interface (CLI) for running analysis
- Child process analysis

## Reporting CLI

- Exporting results (pack and send)
- Text reports: XML, CSV and plain text
- Detect new problems automatically

Create a baseline



Check for regressions

# Automate Regression Analysis

## Command Line Interface

inspxe-cl is the command line:

- **windows:** C:\Program Files\Intel\Inspector XE \bin[32|64]\inspxe-cl.exe
- **Linux:** /opt/intel/inspector\_xe/bin[32|64]/inspxe-cl

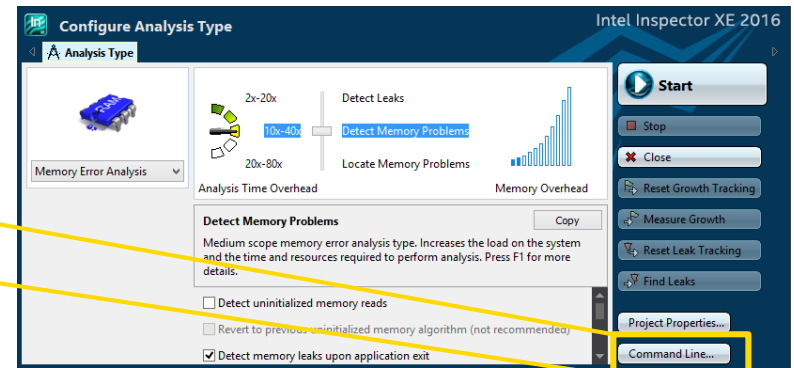
Help:

```
inspxe-cl -help
```

Set up command line with GUI

Command examples:

1. `inspxe-cl -collect-list`
2. `inspxe-cl -collect ti2 -- MyApp.exe`
3. `inspxe-cl -report problems`



# Reporting: regression status

```
inspxe-cl -report status -r r002mi1
```

```
9 problem(s) found  
  2 Investigated  
  7 Not investigated
```

```
Breakdown by state:
```

```
  2 Confirmed  
  4 Not fixed  
  2 Regression  
  1 New
```

# Intel Inspector XE: User APIs

## Enable you to

- Control collection, limit analysis scope
- Specify non-standard synchronization primitives
- Specify custom memory allocation primitives

## To use user APIs:

- Include `ittnotify.h`, located at `<install_dir>/include`
- Insert `__itt_*` notifications in your code
- Link to the `libittnotify.lib` file located at `<install_dir>/<lib32|lib64>`
- Available for C/C++ and Fortran

# Custom memory allocation

```
#include <ittnotify.h>

__itt_heap_function my_allocator;
__itt_heap_function my_reallocator;
__itt_heap_function my_freer;

void* my_malloc(size_t s)
{
    void* p;

    __itt_heap_allocate_begin (my_allocator, s, 0);
    p = user_defined_malloc (s);
    __itt_heap_allocate_end (my_allocator, &p, s, 0);

    return p;
}
... // Do similar markup for custom "realloc" and "free" operations

// Call this init routine before any calls to user defined allocators
void init_itt_calls()
{
    my_allocator = __itt_heap_function_create("my_malloc", "mydomain");
    my_reallocator = __itt_heap_function_create("my_realloc", "mydomain");
    my_freer = __itt_heap_function_create("my_free", "mydomain");
}
```

# Collection control APIs

API	Description
<code>void __itt_suppress_push(     unsigned int etype)</code>	Stop analyzing for errors on the current thread
<code>void __itt_suppress_pop(     void)</code>	Resume analysis
<code>void __itt_suppress_mark_range (     __itt_suppress_mode_t mode,     unsigned int etype,     void * address,     size_t size);</code>	Suppress or unsuppress error detection for the specific memory range (object).
<code>void __itt_suppress_clear_range (     __itt_suppress_mode_t mode,     unsigned int etype,     void * address,     size_t size);</code>	Clear the marked memory range

# User-Defined Synchronization APIs

API	Description
<code>void __itt_sync_acquired (void *addr)</code>	Notify Intel Inspector that synchronization object is acquired by current thread
<code>void __itt_sync_releasing (void *addr)</code>	Notify that the code is about to release the specified synchronization object
<code>void __itt_sync_destroy (void *addr)</code>	Tell the Intel Inspector that the synchronization object will not be used again, so the Intel Inspector can dispose of bookkeeping information associated with this object.

# Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2014, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

## Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804



