# MSRR: Leveraging dynamic measurement for establishing trust in remote attestation

Jason Gevargizian

Information and Telecommunication Technology Center
University of Kansas

*jgevargi@ittc.ku.edu*

April 25, 2019

# Overview

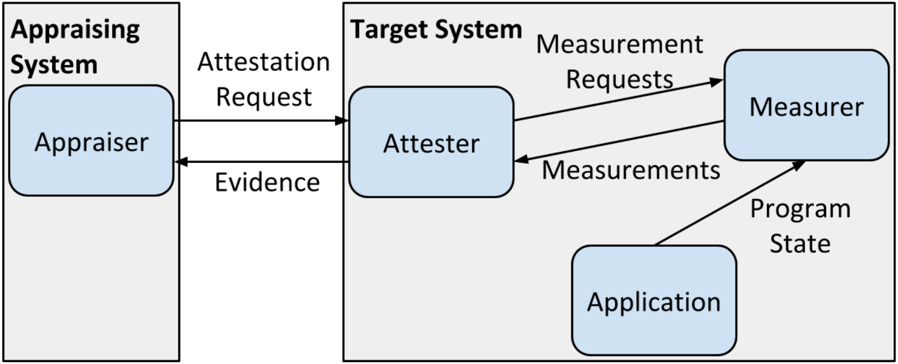
# Introduction, Remote Attestation

- *Remote attestation* is a mechanism for establishing trust
- Needed for communicating entities in distributed computing
- In remote attestation:
  1. `Appraiser` queries `attester` of target system
  2. `Attester` form a proof by invoking `measurers`
  3. `Measurers` collect evidence for proof

**Key Concept: Trust**

Unambiguous identification + Expected behavior compliance − > Trust

# Remote Attestation, Scenario Architecture



# Introduction, Measurement Systems

- **Static measurement**
  - Employed by majority of measurers
  - E.g. **measured boot** = cumulative hash of software binary sequence
  - Does not evidence integrity throughout runtime
- **Dynamic measurement**
  - Sample runtime properties
  - Properties are richer than static hashes
  - Vary greatly from software to software
  - Difficult to measure

# Introduction, Dynamic Measurers

- Must be customized to each application
- Must establish behavioral expectations
- Must specify measurer to evidence expectations
- Customizing measurers is very labarious
- Must analyze source & identify trust critical features
- Burden typically on developer or motivated appraiser
- Cost prohibits widespread adoption of dynamic remote attestation

### Measurement Experts

*An expert must undertake the task of writing measurers. Such a person must have a firm grasp of the purpose and implementation of the target application. Furthermore, they must understand trust, how to evidence trust, and be trained to write good measurers.*

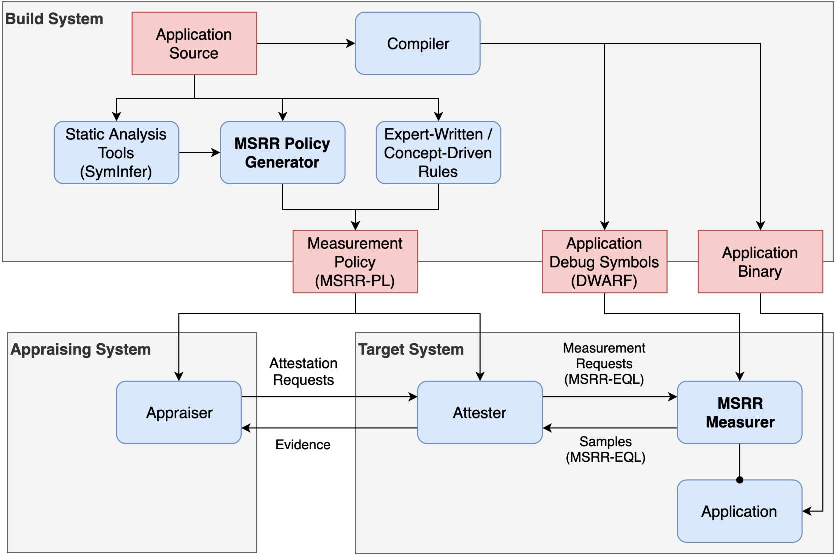
# Introduction, MSRR Approach

- We contribute the MSRR measurement suite
- Techniques to reduce the cost of building measurers
- Make more structured, maintainable, & testable measurers
- Experts no longer need to write measurers from scratch
- Write in efficient high-level policy language
- Leverage automatically generated ‘free’ policies as much as possible

# MSRR, Components

1. **General Purpose Measurer** *(MSRR Measurer)*
   - novel lightweight general purpose measurer
   - provides the common core measurement capabilities
   - low-level querying interface: *MSRR Evidence Querying Language* (MSRR-EQL).
2. **Measurement Policy Language** *(MSRR Policy Language / MSRR-PL)*
   - high-level policy language
   - encapsulates the expected behavior of the target (for appraisal)
   - specifies a sampling schedule (for measurer)
3. **Measurement Policy Generator** *(MSRR Policy Generator)*
   - leverage state of the art static analysis techniques
   - automatically generate MSRR-PL policies
   - automatically configure measurements systems

# MSRR Architecture



# Measurer, General Purpose Measurement System

- MSRR measurement system
- Novel lightweight general-purpose measurer
- Provides core functionality to sample process state
- Attesters invoke measurer via the MSRR Evidence Querying Language (MSRR-EQL)
- Specified by a high-level measurement policy language
    - e.g. MSRR Policy Language (MSRR-PL)

# Measurer, Basic Usage Example

- Example demonstrating attester queries of the MSRR measurer
- Targeting features of a brief C program
- Each attester-measurer exchange in a Scheme-like command-line short form
    - Utilized by MSRR-EQL Interactive Interpreter
- In practice, remote EQL queries are performed over JSON-RPC

# Measurer, Basic Usage Example, Target Program

## Example (simple target application in C)

```c
#include <stdio.h>
#include <unistd.h>

int main() {
  int c = 0;

  while (1) {
    printf("c=%d\n",c);
    c++;
    sleep(1000);
  }
}
```

# Measurer, Basic Usage Example, Exchange 1

*Launch the target executable and attach!*

**Query**

```
(launch_as_target "/path/to/example_binary")
```

**Result**

```
(void)
```

- `Void` result indicates no failure
- Measurer is now attached and ready for sampling

# Measurer, Basic Usage Example, Exchange 2

*Sample the call stack immediately!*

**Query**

```
(measure (callstack))
```

**Result**

```
(sample
  (call_graph_value "main"
    (call_graph_value "sleep"
      (call_graph_value "__nanosleep_nocancel"))))
```

- *On-demand* measurement are served immediately
- Result is one `sample` holding a `call_graph_value`

# Measurer, Basic Usage Example, Exchange 3

*Store a measurement of variable C each time line 8 is reached!*

## Query

```
(hook
 (reach (method_offset_location "main.c" "main" 8) true)
 (action (store (measure (var "c")))))
```

## Result

```
(void)
```

- *Monitoring* measurement are registered for later
- `Hook` associates some event to some action
- `Event` = `reach` of desired instruction
- `Action` = `store` a sampling of $c$

# Measurer, Basic Usage Example, Exchange 4

*Retrieve the stored samples!*

**Query**

```
(retrieve)
```

**Result**

```
(sample_set
  (sample (int_value 33))
  (sample (int_value 34)))
```

- `Sample_set` contains two measurements of $c$
- In practice, the low-level EQL queries are complicated
- Though, EQL queries are produced automatically from MSRR-PL

# Measurer, MSRR-EQL

- MSRR Evidence Query Language (MSRR-EQL)
- Interface for attester to request samples
- Communicated over JSON-RPC
- Specifies **what**, **how**, & **when/where** to sample

## MSRR-EQL Function Modules

Admin and Setup configure measurer; attach/detach target procecesses

Measurement: take samples, store samples, retreive samples

Features: specify properties of target application for sampling

Snapshots: create and manage execution state snapshots of target

Events and Hooks: register and manage monitoring measurements

Locations: specify various code locations for reach events

Control Functions: control flow logic for advanced measurements

| Function | Arguments | Return | Description |
| --- | --- | --- | --- |
| *Admin & Setup* | | | |
| launch_as_target | string | void | Launch executable and attach measurer. |
| release_target | - | void | Detach measurer from target. |
| set_target | string | void | Attach measurer to a process by PID. |
| shut_down | - | void | Terminate measurer. |
| *Measurement* | | | |
| measure | feature | sample | Measure a specific feature of target. |
| retrieve | - | sample_set | Retrieve buffered measurements. |
| store | sample | void | Buffer a measurement for later retrieval. |
| store | string, sample | void | Buffer a measurement with a label. |
| *Feature* | | | |
| callstack | - | feature | Create a *feature* representing the callstack. |
| mem | string, string | feature | Create a *feature* for a memory address with specified format. |
| reg | string | feature | Creates a *feature* for a specific register. |
| var | string | feature | Creates a *feature* for a specific target variable, by source identifier. |
| *Snapshots* | | | |
| disable_auto_snap | - | void | Disable automatic snapping. |
| enable_auto_snap | integer | void | Enable automatic snapping when feature count exceeds threshold. |
| snap | string | void | Create a snapshot of target with given label. |
| to_snap | string, action | * | Evaluate an EQL query on the specified snapshot. |
| *Events & Hooks* | | | |
| action | * | action | Create an object for any expression. |
| delay | integer, boolean | event | Create a timer event with a specified duration. |
| disable | string | void | Disable a given hook by label. |
| enable | string | void | Enable a given hook by labeel. |
| hook | string, event, action | void | Create a hook that evaluates an action when an event occurs. |
| kill | string | void | Kill a given hook by label. |
| reach | location, boolean | event | Create an event that triggers upon target reaching a specified code location. |
| *Locations* | | | |
| file_line_location | string, integer | location | Create a *location* for a file and line number. |
| method_entry_location | string, string | location | Create a *location* for the entry point of a method. |
| method_exit_location | string, string | location | Create a *location* for the exit point of a method. |
| method_offset_location | string, string, integer | location | Create a *location* for a line at an offset from the top of a method. |
| *Control Functions* | | | |
| eq | *, * | boolean | Evaluates the equivalence of the arguments. |
| if | boolean, *, * | * | Evaluate one of two expressions depending upon some condition. |
| not | boolean | boolean | Return the boolean complement of the inputl. |
| seq | *,*, ... | [*, *, ...] | Evaluate a sequence of expressions. |

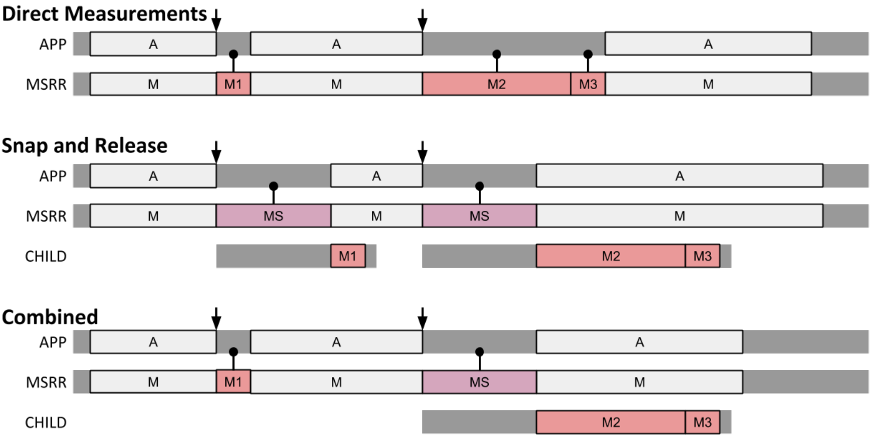
# Measurer, Snapshot Measurements

- *Direct measurements* operate on the target process
- Large requests can impose significant slowdown
- *Snapshot measurements* strategy copies target state
- Measurements queried upon the snapshot itself
- Utilizing Linux *fork* system call
- Automatic snapshot mode uses snap threshold

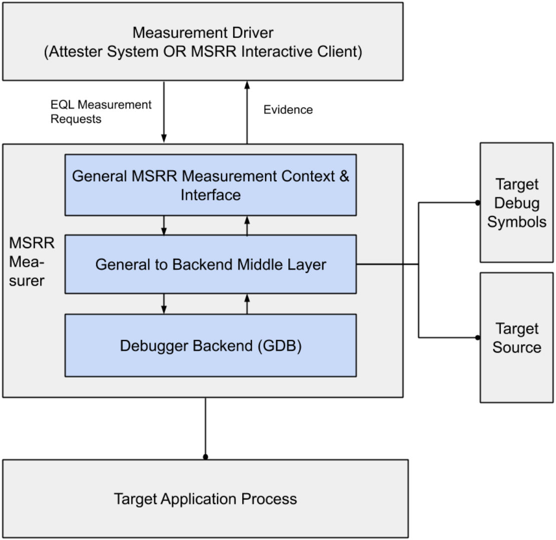
### Fork Implications

Upon a fork, the original process memory is marked *copy-on-write*. Therefore, only the data that is overwritten by the process during sampling needs to be copied.

# Measurer, Snapshot Measurements



# Measurer, Context & Layers



# Policy Language, MSRR-PL

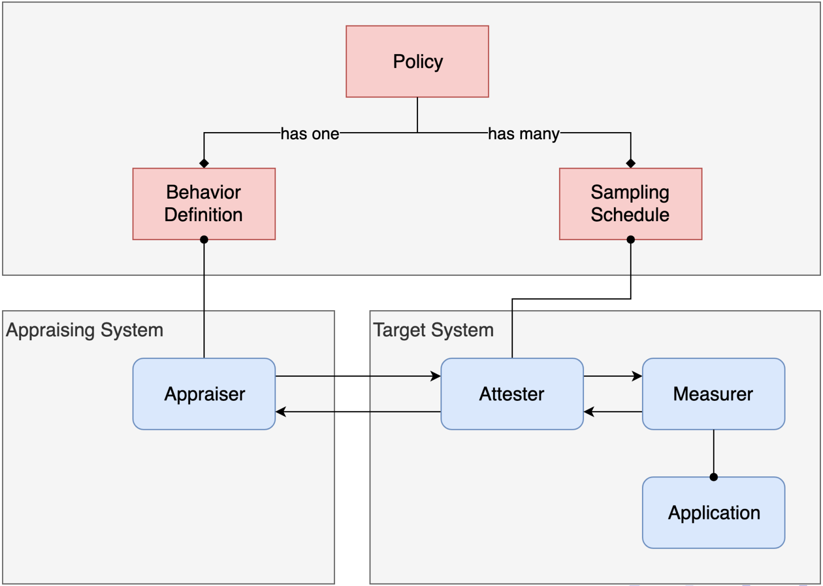
- *MSRR Policy Language* (MSRR-PL) is a high-level policy language
- Write application specific measurement policies for MSRR
- Make the the process of writing measurers structured
- Measurement systems that are more predictable, scalable, and testable.
- Produces to the MSRR-EQL queries

## Main Components

**Expected Behavior Definition** Encapsulate the expected behavior of an application

**Sampling Schedule** Schedule measurement requests which evidence the expected behavior

# Policy Language, Components



# Policy Language, Expected Behavior Definition

- Describes subset of the expected behavior of a target
- Expresses general facts *independent of the measurer*
- Comprised of a set of **rules**
- Rules describe specific properties of the target application
- A policy *has one* expected behavior definition

### Example (Rules in written language)

1. For all instructions, local variable X must be greater than Y.
2. At instruction I1, the local variable *password* must equal 'password123' while local variable *logged_in* equals 'true'.

# Policy Language, Sampling Schedules

- Specify how the measurer should be invoked
- To evidence the expected behavior definition
- Determine how often specific rules should be sampled
- A policy *has many* Sampling Schedules
- Only one schedule can be active at a time

**Example (Multiple Schedules)**

A single expected behavior definition may be associated with two schedules: one that samples for each rule at a moderate frequency and another that only measures one rule, yet does so very frequently.

# Policy Language, Example

- Let's write a simple policy
- For this simple C program
- Observe that $x$ is incremented by two
- Observe $x$ should always be even
- Let's encapsulate the evenness of $x$ in a policy

**Example (Target Program)**

```
#include <stdio.h>
#include <unistd.h>

int main() {
  int x = 2;

  while (1) {
    printf("x=%d\n",x);
    x+=2;
    sleep(3);
  }
}
```

# Policy Language, Example, Validation Function

- Start with the expected behavior definition
- We need one rule with a definition of evenness

**Example (Definition of evenness in C)**

```
bool is_even( int x ) {
    return x % 2 == 0;
});
```

- `ValidationFunction` are used in MSRR-PL
- Essentially a lambda of type `SampleSet` $->$ `bool`
- `SampleSet` is a collection of `Samples`
- `Samples` hold data taken by measurer

# Policy Language, Example, Validation Function

## Example (Validation Function)

```
Policy policy;

policy.behavior_definition
  .validation_functions["is_even_validation_function"] =
    new ValidationFunction(
      [](SampleSet samples) {
        int x = samples.getAsInt("x_parameter");
        return x % 2 == 0;
      }
    );
```

# Policy Language, Example, Features

- Start constructing the parameter for validation function
- Declare a `Feature` for $x$ using the its source identifier

**Example (Feature)**

```
policy.behavior_definition.features["feature_x"] =
  new VariableFeature("x");
```

# Policy Language, Example, Locations

- Specify where $x$ shall be even
- Using a `Location` scope
- This `FileRangeLocation` scope captures the body of the loop

**Example (Feature)**

```
policy.behavior_definition.locations["loop_body_location"] =
  new FileLineRangeLocation("main.c", 8, 10);
```

# Policy Language, Example, Occurrences

- Specify when $x$ shall be even
- Defining an `Occurrence` scope for the `Location`
- $x$ should always even at our location
- We define an `OriginOccurrence` scope
- Origin occurrences are unbounded
- Used as a point of reference for other occurrence scopes

**Example (Feature)**

```
policy.behavior_definition
  .occurrences["every_loop_occurrence"] =
    new OriginOccurrence("loop_body_location");
```

# Policy Language, Example, Rules

- Last step of the expected behavior definition
- Define the evenness rule
- With a scoped `Parameter`
- `Parameter` is our `Feature` scoped to our `Occurence`
- (which in turn scopes to the associated `Location`)
- Rule associates our one parameter to the validation function

## Example (Feature)

```
policy.behavior_definition.parameters["x_parameter"] =
  new Parameter("x_feature", "every_loop_occurrence");

policy.behavior_definition.rules["is_even_rule"] =
  new Rule("is_even_validation_function", {"x_parameter"});
```

# Policy Language, Example, Sampling Schedule

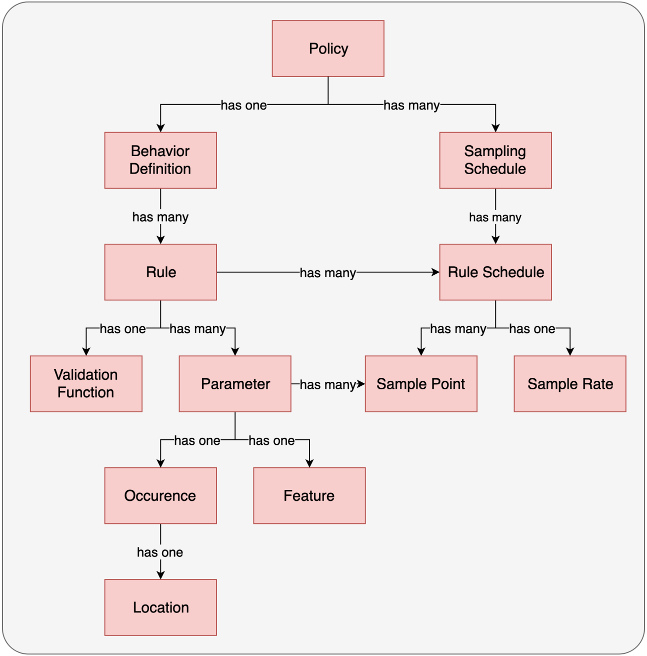
- Policy needs at least one sampling schedule
- Our schedule will:
  - Take a single sample every other iteration of the loop
  - At a random instruction in the loop body
- We define a new `SamplingSchedule`
- Using `SampleFrequency` subtype `EveryOtherIteration`
- We add a `RuleSchedule` for the *evenness* rule
- Using the `SamplePoint` subtype `RandomLineSamplePoint`

# Policy Language, Example, Sampling Schedule

## Example (Feature)

```
policy.sampling_schedules["default_schedule"] =
  new SampleSchedule();

policy.sampling_schedules["default_schedule"]
  .rule_schedules["is_even_rule_schedule"] =
    new RuleSchedule(
      "is_even_rule", EveryOtherIteration(),
      {RandomLineSamplePoint()}
    );
```

```
Policy policy;

policy.behavior_definition
  .validation_functions["is_even_validation_function"] =
    new ValidationFunction(
      [](SampleSet samples) {
        int x = samples.getAsInt("x_parameter");
        return x % 2 == 0;
      }
    );

policy.behavior_definition.features["feature_x"] =
  new VariableFeature("x");

policy.behavior_definition.locations["loop_body_location"] =
  new FileLineRangeLocation("main.c", 8, 10);

policy.behavior_definition
  .occurrences["every_loop_occurrence"] =
    new OriginOccurrence("loop_body_location");

policy.behavior_definition.parameters["x_parameter"] =
  new Parameter("x_feature", "every_loop_occurrence");

policy.behavior_definition.rules["is_even_rule"] =
  new Rule("is_even_validation_function", {"x_parameter"});

policy.sampling_schedules["default_schedule"] =
  new SampleSchedule();

policy.sampling_schedules["default_schedule"]
  .rule_schedules["is_even_rule_schedule"] =
    new RuleSchedule(
      "is_even_rule", EveryOtherIteration(),
      {RandomLineSamplePoint()}
    );
```


Policy
has one
has many
Behavior Definition
Sampling Schedule
has many
has many
Rule
has many
Rule Schedule
has one
has many
has many
has one
Validation Function
Parameter
has many
Sample Point
Sample Rate
has one
has one
Occurence
Feature
has one
Location

# Policy Language, Validation Functions

- `ValidationFunction` contains a lambda
- Type `SampleSet` $->$ `boolean`
- Samples = actual measurements taken of various features
- boolean output indicates pass or fail

**Example (Validation Function)**

```
policy.behavior_definition
  .validation_functions["is_positive"] =
    new ValidationFunction(
      [](SampleSet samples) {
        int num = samples.getAsInt("num_parameter");
        return x > 0;
      }
    );
```

# Policy Language, Location Scopes

- `Locations` are sed to restrict a `Parameter` of a `Rule`
- To some code region(s)
- Basic location types
- Set operation types

**Example (Feature)**

```
policy.behavior_definition.locations["foo_location"] =
  new FileMethodLocation("main.c", "foo");
```

# Policy Language, Location Scopes

## Basic Types

**FileClassLocation (F, C)** All instructions that are part of class C of file F.

**FileMethodLocation (F, M)** All instructions that are part of method M of file F.

**FileRangeLocation (F, I, J)** All instructions that exist between line numbers I and J of file F.

**FileLineLocation (F, I)** Instruction at line I of file F.

# Policy Language, Location Scopes

## Set Operation Types

UnionLocation (L1, L2) The union of all instructions of locations L1 and L2.

IntersectionLocation (L1, L2) The intersection of all instructions of locations L1 and L2.

DifferenceLocation (L1, L2) The difference of all instructions of locations L1 and L2.

SymmetricDifferenceLocation (L1, L2) The symmetric difference of all instructions of locations L1 and L2.

# Policy Language, Occurrence Scopes

- `Occurrences` used with a `Location`
- Bound a `Parameter` to a relative time at specified location
- Defined relative to each other
- `OriginOccurrence` - default - unbounded point of origin

## Example (Next Occurence)

```
policy.behavior_definition
  .occurrences["l1_occurrence"] =
    new OriginOccurrence("location_1");

policy.behavior_definition
  .occurrences["l2_after_l1_occurrence"] =
    new NextOccurrence("location_2", "l1_occurrence");
```

# Policy Language, Occurrence Scopes

## Occurrence Types

OriginOccurrence (L) Any occurrence of location L. Serves as a point of origin for other occurrences.

NextOccurrence (L, O) The immediate next occurrence of Location L after the Occurrence O.

KthNextOccurrence (L, O, k) The k-th occurrence of location L after the Occurrence O.

FirstOccurrence (L) The *absolute* first occurrence of location L.

# Policy Language, Sample Rates

Specify how often to sample `Parameters`

## Sampling Rate Types

- **EveryIteration** All matching iterations of the associated scoped parameter is sampled.
- **EveryOtherIteration** Every other iteration of the associated scoped parameter is sampled.
- **EveryKthIteration (k)** Every k-th iteration of the associated scoped parameter is sampled.
- **EveryIterationAfterDelay (d)** Each iteration after duration `d` has expired.
- **ChanceOfSampling (p)** Each iteration has a `p` percent chance of sampling.
- **SkipSampling** No iterations are sampled. Rule is disabled.

# Policy Language, Sample Points

Specify where to sample with `Location`

## Sample Point Types

**FileLineSamplePoint (F, L)** Sample at line L of file F.

**FirstLineSamplePoint** Sample at the first line of the associated location scope.

**KthLineSamplePoint (K)** Sample at the k-th line of the associated location scope.

**LastLineSamplePoint** Sample at the last line of the associated location scope.

**RandomLineSamplePoint** Sample at a random line in the location scope.

**MethodEntrySamplePoint (M)** Sample at the entry to method M.

**MethodExitSamplePoint (M)** Sample at the exit of method M.

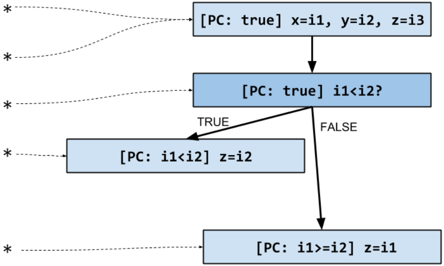
# Generator, Measurement Policy Generation

- Builds upon the MSRR Measurer and MSRR Policy Language
- Technique to automate the generation of measurement policies
- The process of producing tailored measurement systems
- For some cases: eliminate manual effort
- For the rest: augment manual policies
- Expert effort on only most critical apps and their structures

# Generator, SymInfer, KLEE, & DIG

- SymInfer employs *symbolic execution* to produce program invariants
- Symbolic execution is a type of program execution
- *Symbolic values* instead of concrete values
- All paths explored instead of one

```
int x,y,z;

...

if (x < y) {

   z = y;

} else {

   z = x;

}
```



# Generator, Invariant Format

## Example (Invariant Format)

```
*** programs/nla/cohendiv.c, 2 locs, invs 13 (4 eqts),
  inps 187, time 300.355239153 s, rand 71:
25: a*y - b == 0, q*y + r - x == 0, -b <= -1, b - r <= 0,
  r - x <= 0, -y <= -1
37: a*y - b == 0, q*y + r - x == 0, -a <= 0, r - y <= -1,
  -a - r <= -1, -r <= 0, a - q <= 0
```

# Generator, Validation Function

## Example (Validation Function)

```
policy.behavior_definition
  .validation_functions["validation_function_1"] =
    new ValidationFunction(
      [](SampleSet samples) {
        int a = samples.getAsInt("a");
        int b = samples.getAsInt("b");
        int q = samples.getAsInt("q");
        int r = samples.getAsInt("r");
        int x = samples.getAsInt("x");
        int y = samples.getAsInt("y");
        return a*y - b == 0 && q*y + r - x == 0 &&
          -b <= -1 && b - r <= 0 && r - x <= 0 &&
          -y <= -1;
        }
      );
```

# Experiment Specifications

- System running 64-bit Fedora 24 with 32 GB of memory
- Quad-core Intel Xeon 1.8 Ghz processor
- Benchmarks:
  1. Custom micro-benchmarks
  2. Non-Linear Arithmetic (NLA) micro-benchmark suite
  3. SPEC CPU 2006 benchmark suite with the reference data sets
- Relevant benchmarks compiled with the -g option to produce the DWARF symbols

# Experiment 1

- SPEC CPU 2006 benchmarks
- MSRR measurer overhead with no measurement
- Attach to the target application and wait indefinitely
- No discernible overhead that is within the margin of error

## Key Takeaways

- MSRR measurer attachment to target has negligible overhead

# Experiment 2

- Simple micro-benchmark (computing the Fibonacci sequence)
- Measure the cost of individual MSRR-EQL features
- Collect approximately 22,000 samples. `snap` every 10,000 msec.
- `callstack`, `reg`, `mem`, `hook`, and `snap` events have an overhead of 0.54 msec, 0.32 msec, 0.32m sec, 1.94 msec, 96.45 msec

## Key Takeaways

- Individual measurements have low overhead
- Some (`callstack` and `snap`) depend on stack and memory usage
- Suggested snap threshold in the range of 200-300

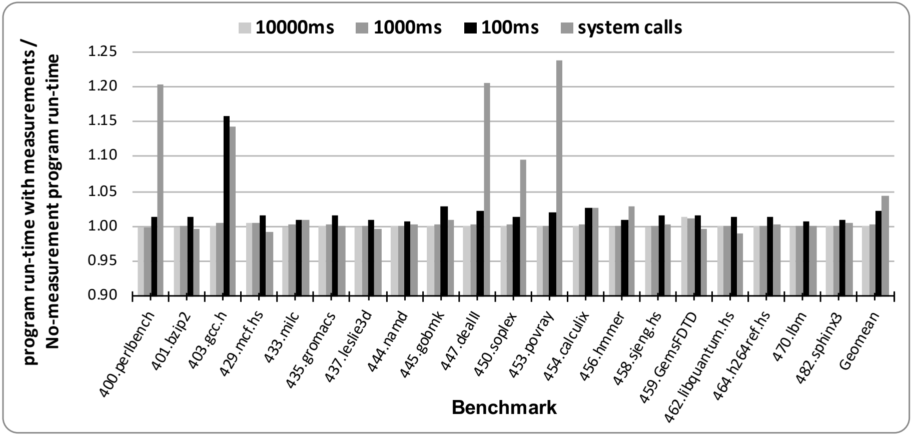
# Experiment 3

- SPEC CPU 2006 benchmarks
- Overhead imposed when sampling at different measurement frequencies
- Periods: 100 msec, 1000 msec, 10,000 msec, and at every system call
- Overhead of 0.08%, 0.25%, 2.14%, and 7.95% for call-stack measurements taken every 10,000 msec, 1000 msec, 100 msec, and at all system calls
- Standard deviations were small relative to their means

## Key Takeaways

- MSRR overheads are low for even high degrees of measurement
- Trade-off between performance & accuracy of trust inferences
- 403.gcc was 115.7 because of very large call stacks

# Experiment 3



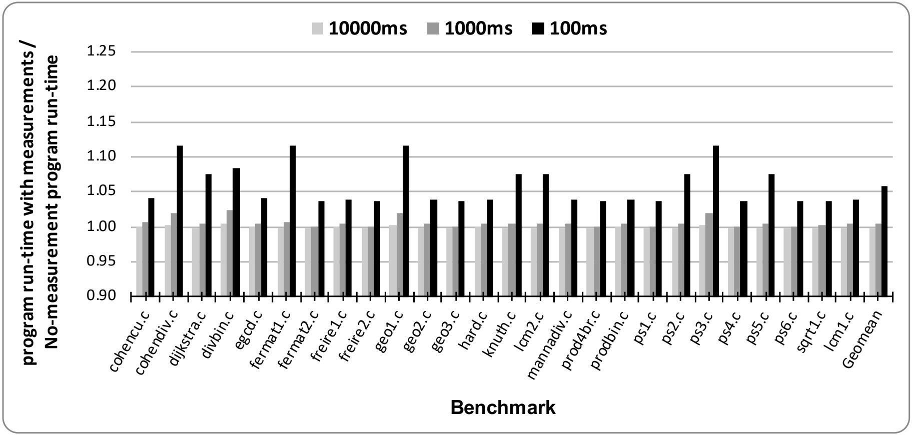
# Experiment 4

- Non-Linear Arithmetic (NLA) micro-benchmark suite
- Automatically generated MSRR-PL policies
- Sampling periods of 100 msec, 1000 msec, and 10,000 msec
- Overhead of 0.53% and 5.29% for call-stack measurements taken every 1000 msec and 100 msec
- Overhead at 10,000 msec was statistically insignificant
- Average standard deviation was 0.67%
- All standard deviations fell in the range of 0.13% and 3.51%

## Key Takeaways

- Automatically generated MSRR-PL policies produce low-overhead measurers
  - For both lax and taxing sampling schedules
- Many types of measurements will tend to have negligible overhead
  - Most measurements with occurrence periods on the order of seconds
  - E.g. Those involving human interaction

# Experiment 4



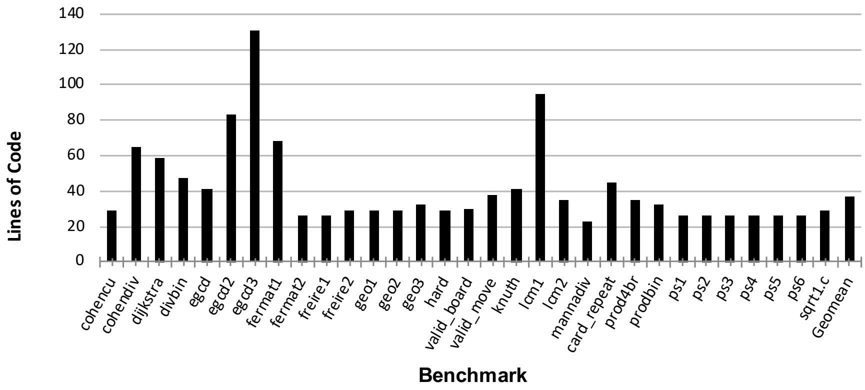
# Experiment 5

- Produced several representative policies:
  - bluffinmuffin, a Texas Hold'em card game simulator
  - 27 MSRR auto-generated NLA policies
  - Two policies for the DreamChess program
- Report code metrics: lines of code, token count, cyclomatic complexity number
- 36.9 lines of code and 505.8 tokens on average for all policies
- Lines of code and token count scaled linearly with number of params
- Mean CCN was 3.14 at a per-method average and 4.85 at a per-method maximum

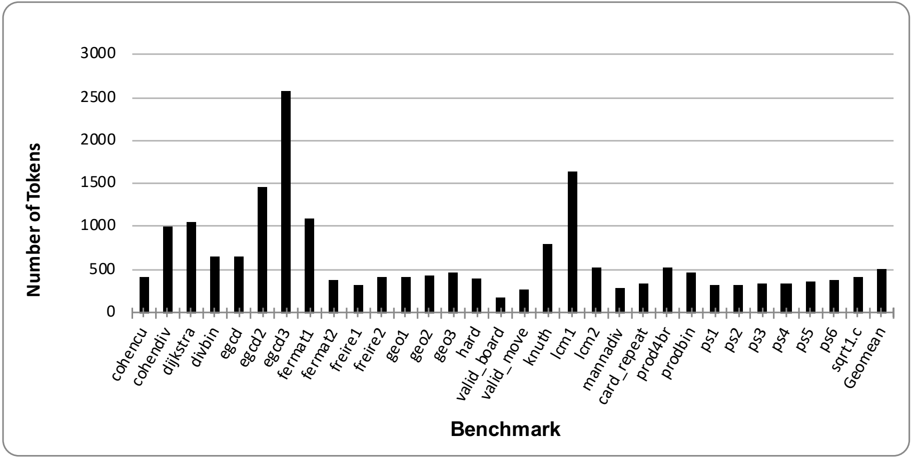
## Key Takeaways

- Automatic and manual polices generally have low complexity
- CCNs were well below *McCabe's original suggested limit of 10*
- Complexity depends on property, little is introduced by MSRR-PL
- Optimizations for size and dev time in syntax aids and templating

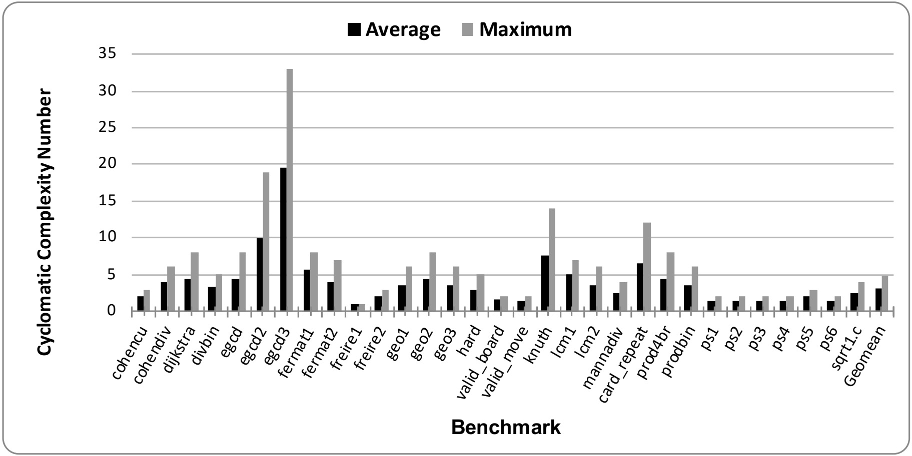
# Experiment 5, Lines of Code



# Experiment 5, Tokens Count

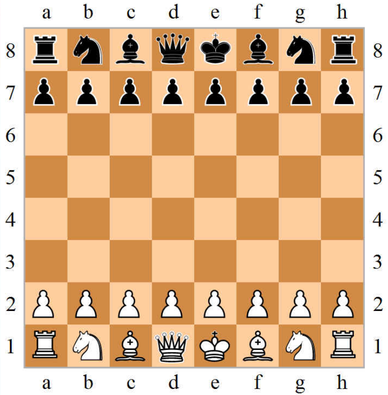


# Experiment 5, Cylcomatic Complexity



# DreamChess Case Study

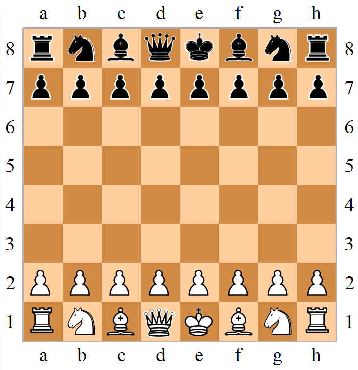
- MSRR applied to DreamChess chess game simulator
- Appraiser acting on behalf of 'gaming authority' or 'referee'
- Goal is to verify legal games
- Money, prestigious chess titles, gaming provider credentials at stake
- Let's develop a measurer to validate legal chess moves



# DreamChess, Code Snippets

```
#define WHITE_PAWN 0
#define BLACK_PAWN 1
#define WHITE_KNIGHT 2
#define BLACK_KNIGHT 3
#define WHITE_BISHOP 4
#define BLACK_BISHOP 5
#define WHITE_ROOK 6
#define BLACK_ROOK 7
#define WHITE_QUEEN 8
#define BLACK_QUEEN 9
#define WHITE_KING 10
#define BLACK_KING 11
#define NONE 12
```

```
typedef struct board
{
  int turn;
  int square[64];
  int captured[10];
  int state;
} board_t;
```



# DreamChess, Validation Function

## Example (Validation Function)

```
policy.behavior_definition
  .validation_functions["move_validation_function"] =
    new ValidationFunction(
      [](SampleSet samples) {
        vector<int> squares_initial =
          samples.getAsVector<int>("initial_board");
        vector<int> squares_final =
          samples.getAsVector<int>("successor_board");
        vector<int> squares_difference =
          subtract_vectors(squares_initial, squares_final);
        return count_nonzero(squares_difference)==2;
      });
```

# DreamChess, Features & Scopes

## Example (Feature & Scopes)

```
policy.behavior_definition.features["squares_feature"] =
  new VariableFeature("board->square");

policy.behavior_definition.locations["make_move_location"] =
  new FileMethodLocation("board.c", "make_move");

policy.behavior_definition
  .occurrences["initial_occurrence"] =
    new OriginOccurrence("make_move_location");

policy.behavior_definition
  .occurrences["successor_occurrence"] =
    new NextOccurrence(
      "make_move_location", "initial_occurrence"
    );
```

# DreamChess, Parameters & Rule

## Example (Parameter & Rule)

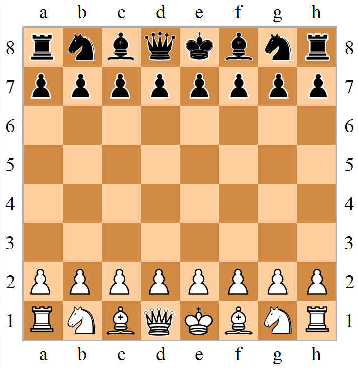
```
policy.behavior_definition
  .parameters["initial_board"] =
    new Parameter("squares_feature", "initial_occurrence");

policy.behavior_definition
  .parameters["successor_board"] =
    new Parameter(
      "squares_feature", "successor_occurrence"
    );

policy.behavior_definition.rules["valid_move_rule"] =
  new Rule(
    "move_validation_function",
    {"initial_board", "successor_board"}
  );
```

# DreamChess, Sampling Schedule

## Example (Sampling Schedule)

```
policy.sampling_schedules["default_schedule"] =
  new SampleSchedule();

policy.sample_schedules["default_schedule"]
  .rule_schedules["valid_move_rule"] =
    new RuleSchedule(
      "valid_move_rule", EveryIteration(),
      {FirstLineSamplePoint(), FirstLineSamplePoint()}
    );
```

```
(sample_set
  (sample (int_value
    6 2 4 8 10 4 2 6
    0 0 0 0 0 0 0 0
    12 12 12 12 12 12 12 12
    12 12 12 12 12 12 12 12
    12 12 12 12 12 12 12 12
    12 12 12 12 12 12 12 12
    1 1 1 1 1 1 1 1
    7 3 5 9 11 5 3 7
  ))
  (sample (int_value
    6 2 4 8 10 4 2 6
    0 0 0 0 12 0 0 0
    12 12 12 12 12 12 12 12
    12 12 12 12 0 12 12 12
    12 12 12 12 12 12 12 12
    12 12 12 12 12 12 12 12
    1 1 1 1 1 1 1 1
    7 3 5 9 11 5 3 7
)))
```



# Final Thoughts

- Explore techniques to spend less energy to build higher quality measurers
- MSRR Measurer eliminates the need redevelop core measurement functionality
- MSRR-PL expedites and systematizes the writing of per-application measurers
- MSRR generator demonstrates how static analysis tools can produce policy coverage very cheaply

# Questions?