MSRR: Leveraging dynamic measurement for establishing trust in remote attestation

Jason Gevargizian

Submitted to the Department of Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Dissertation Committee:

Dr. Prasad Kulkarni, Chairperson

Dr. Arvin Agah

Dr. Bo Luo

Dr. Kevin Leonard

Dr. Perry Alexander

Date Defended

The Dissertation Committee for Jason M. Gevargizian certifies
that this is the approved version of the following dissertation:

MSRR: Leveraging dynamic measurement for establishing trust in remote attestation

Prasad A. Kulkarni, Chairperson

Date Approved

Abstract

Measurers are critical to a remote attestation (RA) system to verify the integrity of a remote untrusted host. Runtime measurers in a dynamic RA system sample the dynamic program state of the host to form evidence in order to establish trust by a remote system (*appraisal system*). However, existing runtime measurers are tightly integrated with specific software. Such measurers need to be generated anew for each software, which is a manual process that is both challenging and tedious.

In this paper we present a novel approach to decouple application-specific measurement policies from the measurers tasked with performing the actual runtime measurement. We describe the *MSRR (MeaSeReR) Measurement Suite*, a system of tools designed with the primary goal of reducing the high degree of manual effort required to produce measurement solutions at a per application basis.

The MSRR suite prototypes a novel general-purpose measurement system, the MSRR Measurement System, that is agnostic of the target application. Furthermore, we describe a robust high-level measurement policy language, *MSRR-PL*, that can be used to write per application policies for the MSRR Measurer. Finally, we provide a tool to automatically generate MSRR-PL policies for target applications by leveraging state of the art static analysis tools.

In this work, we show how the MSRR suite can be used to significantly reduce the time and effort spent on designing measurers anew for each application. We describe MSRR's robust querying language, which allows the appraisal system to accurately specify the what, when, and how to measure. We describe the capabilities and the limitations of our measurement policy generation tool. We evaluate MSRR's overhead and demonstrate its functionality by employing real-world case studies. We show that MSRR has an acceptable overhead on a host of applications with various measurement workloads.

Acknowledgements

I want to extend my gratitude to my advisor, Prasad Kulkarni, for his support, his mentorship, and his patience during my time at the University of Kansas. He is sharp, kind, an excellent instructor, and I am very fortunate to have had the opportunity to work with him as a graduate student.

I also wish to extend my thanks to my family:

To my dad, Benjamin, who I miss dearly. The way that he pursued his interests with unparalleled determination and deftness has stuck with me. I often find myself considering what he would do, were he in my shoes, to find strength in the toughest of times.

To my brother and sister, David & Jill, who I have become close with during my time here as a graduate student. They are my most cherished friends and their support has meant everything.

To my mother, Deidra: Mom, no person has put the time and energy into my life that you have, not by a long shot. Thank you for wishing the world of me. By the way, Friday evening breaks puzzling over the New York Times crossword with you and the family have really made even the toughest weeks of graduate school something special.

Lastly, I would like to extend my thanks to all my good friends and colleagues who have supported me all along the way. Thank you.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

An entity is said to be *trusted*, or *trustworthy*, from a point of reference, if it meets two criteria: (1) the entity can be unambiguously identified, from the point of reference and (2) the entity can be observed, from the reference point, as behaving in accordance with previously known expectations [30, 34].

Remote attestation provides mechanisms for untrusted entities to prove their integrity to a remote party. Such mechanisms are integral for communicating entities to establish trust in a distributed computing environment [20].

In remote attestation, an *appraisal system* seeks to establish trust of a *target* by requesting *evidence*. An *attestation system* (attester) on the target entity answers this request by invoking *measurement systems* (measurers) to gather evidence from the relevant features of the target entity. The attestation system takes the gathered evidence and compiles it into an *attestation*, or proof, as to the facts of interest to the appraising entity [10, 11].

Evidence can take many forms, and it can offer critical insight to a wide array of properties about an entity of interest. Often, evidence will include static and configuration information, runtime measurements, and even so-called metaevidence. What evidence is relevant, when, and under what conditions it shall be sufficient to establish trust, depends on the nature of the appraisal system's inquiry and the expediency of the trust-contingent interactions to follow.

Most remote attestation techniques provide integrity evidence by only measuring the *static* properties of the target hardware/software system [49]. Static software properties typically include a cumulative hash constructed during the *measured boot* process, and other static state of the running software such as code regions. However, programs from trusted vendors are still vulnerable to runtime security attacks, such as buffer overflow attacks. *Static* remote attestation techniques cannot measure the integrity of the software state that can dynamically change after the programs begin execution.

Dynamic remote attestation attempts to remedy this limitation with static remote attestation based systems [24, 28]. These dynamic remote attestation approaches measure runtime properties of the executing software, often expressed as boolean propositions that must be true during a normal execution of the program, and are specific to each program. Measurement policies for dynamic remote attestation reason about the properties and relationships between dynamically varying program state that is held in architecture registers, structures on the call stack, and objects on the stack, heap, or the global region.

Runtime properties vary greatly from software to software. Even in cases where applications are similar in purpose, for example two different virus checkers, attestation critical structures and program variables, and their properties and relationships, may differ significantly. As such, measurement functionality must be tailored to each critical application on the target entity.

Existing runtime measurement systems are commonly built with their target applications in mind. These measurement systems are very brittle and cannot be utilized to gather evidence from any other target softwares. This limitation includes, often, different releases of the same application [28], because even small refactors or extensions to said software can lead to changes in trust critical features, their locations, windows of measurement, expected states, et cetera.

Establishing behavioral expectations and customizing measurement systems to gather meaningful data to evidence said expectations is difficult. The process requires an expert, typically the developer or a motivated appraiser, to analyze the application's source in order to detail program behavioral expectations critical for establishing trust and to identify critical program structures and variables that can be sampled to form evidence. Such an expert must be (a) well informed in the domain of the target application and its expected behavior and (b) trained to be able to write quality measurement systems in order to create an accurate and efficient measurer to collect evidence for said application.

1.1 Problem Definition

We believe that the high cost of building customized measurement systems for user-level applications prohibits widespread adoption of dynamic remote attestation in trusted computing.

As it stands, the tailoring of measurement systems to each new target application is very difficult. Specifically, the process of determining trust-critical expected behavior for a target application is manual, requires the attention of an expert, and is consequently preventatively expensive. Furthermore, the process of writing the measurement systems to actually gather the evidence needed, at runtime, to establish the aforementioned expected behavior also is difficult and requiring of such an expert.

As it stands, there is not a single general purpose measurement system with a well-defined and common evidence querying interface that exists for native applications. There is no unified way to be able to easily write measurement policies at a high level, in order to simplify the life of the expert. There are no tools, static analysis or otherwise, to automatically generate program expectations that can be readily supplied to a measurement system for the purposes of automating, completely or partially, the process of writing the per application measurement policies.

The problem this paper seeks to address is that lack of framework, infrastructure, and automatic tools in the current state of the art remote attestation and measurement technologies. With the functionality of these desired missing components combined, the ability to automatically generate a full dynamic attestation system is afforded. And, consequently, manual effort by niche experts could be significantly reduced and, in many cases, eliminated.

1.2 Contributions

We contribute the MSRR Measurement Suite. The MSRR measurement suite is a set of tools to designed to eliminate and reduce the manual effort required by an expert in the process of building per application measurers.

The MSRR methodology focuses on the decoupling of program-specific policies, *measurement policies*, from the measurement system itself. As such, the low-level mechanics common to measurement systems are provided and made accessible via the MSRR Evidence Querying Language (MSRR-EQL). In this way, measurement system writers need only to spend effort on the components of the measurement system that actually vary from software to software: id est, the measurement policies.

The MSRR method provides an a clear and concise high-level measurement policy language, MSRR Policy Language (MSRR-PL), to further reduce the manual effort required by measurement writers. This policy language allows one to capture both the expected behavior of the target application and an associated sampling schedule to drive the actual sampling by the relevant measurement system.

The MSRR suite also provides a tool to automatically generate program policy rules utilizing state of the art static analysis techniques. This allows one to achieve a layer of rule coverage for any set of applications, automatically and for very little cost. In many cases, this can eliminate the need for an expert entirely. In the more demanding or challenging cases, this automatic technique can be used in tandem with the expert to greatly reduce the expert's scope of work.

The notable features of the MSRR Measurement Suite are as follows:

1. **General Purpose Measurement System**

   The *MSRR Measurement System* is a novel lightweight general purpose measurer. This measurement system provides the core measurement capabilities common to all measurers and makes them accessible via a low-level querying language, the *MSRR Evidence Querying Language* (MSRR-EQL).

2. **Measurement Policy Language**

   The *MSRR Policy Language* (MSRR-PL) is a high-level policy language that can be leveraged to efficiently write application specific measurement policies for MSRR. The Measurement Policy language allows one to easily describe what is to be measured and when. Furthermore, the policy language specifies the validation criteria for subsequent appraisal; that is, the logic that defines the expected values and relationships of trust critical features. A policy encapsulates both the expected behavior definition of the target application and a sampling schedule to drive the actual sampling by the measurement system.

3. **Measurement Policy Generation**

   The *MSRR Policy Generator* is a tool to leverage state of the art static analysis techniques which find program invariants in order to automatically generate policies and configure remote attestation systems. This tool can be used to generate complete policies from scratch, and it can be used in tandem with manual policies written by an expert

The techniques we have employed in our prototypes can be used to significantly reduce the time and effort required to develop application specific runtime measurers. Using the full suite, a complete measurement system can be generated automatically. In cases where more coverage is required, the automatic policies generated by the MSRR Policy Generator can be augmented with manually policies. To this end, the MSRR Policy Language provides the ability to write such augmentations easily.

1.3 Outline

The remainder of this dissertation is organized as follows. We begin by presenting background material in chapter 2. In chapter 3, we describe MSRR Measurement System, its capabilities, and its MSRR Evidence Querying Language interface. In chapter 4, we describe the MSRR Policy Language, its features, and provide real policy examples. In chapter 5, we describe the MSRR Policy Generator, its usage, and its limitations. In chapter 6, we demonstrate the performance of our suite with a series of experiments and subsequent analyses. In chapter 7, we demonstrate our suite's expressiveness with more real world examples and case studies. In chapter 8, we discuss future work. In chapter 9, we summarize our contributions.

Chapter 2

Background & Related Work

In this chapter we present the background for this work and other related approaches for remote attestation. We will discuss trust, remote attestation, the role of measurement in remote attestation, and the various types of measurement systems.

2.1 Trust and Remote Attestation

Trust is a critical ingredient to the effective and safe communication of various entities, each with varied agendas, features, and quite often vulnerabilities. In order to establish trust, one must prove its own integrity by establishing its identity to the observer. Once one's identity has been strongly and unambiguously confirmed, one must then prove that it is behaving in accordance with some previously known set of expectations, as determined by its alleged hardware and software configurations [30].

Remote attestation is a mechanism that provides the process for untrusted entities to prove their integrity to a remote party, or appraising entity. These such mechanisms are essential to establish trust in the increasingly diverse and distributed computing environments [20].

As such, remote attestation can be considered a *fault tolerance* technique. Fault tolerance techniques focus on failure detection, mitigation of said failures, and further failure avoidance.[2, 3]

During remote attestation, an *appraisal system* (appraiser) seeks to establish trust of a *target* entity by requesting information about its hardware and software configurations. This information serves as *evidence* in the context of the *attestation*, or proof, to establish trust. An *attestation system* (attester) on the target entity is charged with compiling the evidence and returning it to the appraiser. The attester facilitates this task by invoking *measurement systems* (measurers) to gather the evidence from the relevant features of the target entity. Finally, the appraiser receives the attestation response back from the attester and determines wether or not the evidence satisfies the expectations set forth prior to the remote attestation process.

2.1.1 Components of a Remote Attestation Scenario

The key components in a typical remote attestation scenario and their interactions are as follows:

- **Appraising Entity / Point of Reference**

  The appraising entity seeks to establish trust in another entity.

  - **Appraisal System**

    The appraisal system is a piece of software that resides on the appraising entity that seeks to make an *appraisal* of some other entity. An appraisal is to evaluate the trustworthiness of an entity by requesting and subsequently examining evidence from a remote machine. The appraisal system has with it a set of criteria to describe the expected trustworthy behavior, aka the *expected behaviors*, of the target entity, against which it will evaluate the evidence.

- **Target Entity**

  The target entity is the subject of an appraisal which seeks to prove its trustworthiness to the appraising entity.

  - **Attestation System (Attester)**

    The attestation system, or attester, is a piece of software that resides on the target entity. The attester communicates with the appraisal system in order to establish trust with the appraising entity. Specifically, it receives requests for evidence from remote appraisal systems and responds with an attestation. The attestation is a proof which comprises relevant evidence for the given appraisal.

    To facilitate this task, the attester queries local measurement systems to gather evidence from various features of the target entity. In some cases, the attester must make nested attestation requests of other dependent or related systems to which parts of its evidence are dependent.

  - **Measurement System (Measurer)**

    The measurement system is a piece of software that resides on the target entity which collects measurements of trust critical features, including but not limited to user-space applications, as requested by the local attester. The measurements take the form of data *samples* and, in the context of a remote attestation, they function as *evidence*, either evidence of trustworthiness or otherwise.

- **Target Application**

  The target application is any piece of software on the target entity that is either directly or indirectly of interest to the remote appraisal system in establishing trust. This software piece is the subject of measurement by the measurement system(s).

  Though, measurement systems can sample other aspects of an entity, runtime application measurers, as discussed later in this chapter, are the primary concern of this paper.

By way of remote attestation, to establish trust, an appraisal system must first establish that the target is indeed the entity it intends to appraise. Strong identification unambiguously binds an identity to a target entity and any subsequent information that said entity provides. Second, an appraisal system must determine that the target entity is running software that satisfies previously known expectations. Furthermore, it must be established that said software is running on hardware that is within expectations. Going forward, the appraisal system must continually establish that the aforementioned components have not become corrupted or compromised, either erroneously or as the result of a deliberate attack.

As such, evidence must confirm the initial configuration and continually confirm trust in subsequent running state throughout the life of the desired trust relationship between appraising entity and the target entity. Furthermore, evidence also must confirm the measurement and attestation systems with so-called *metaevidence.*

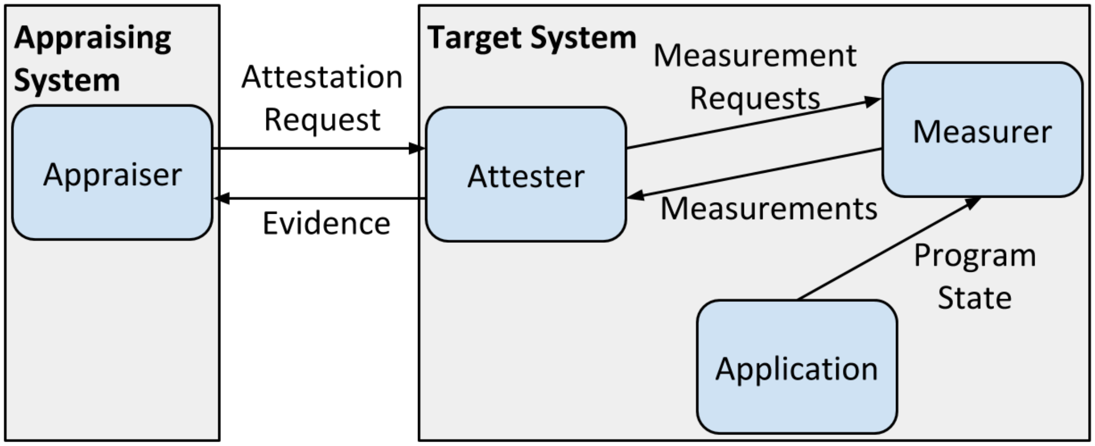


Figure 2.1. Architecture of a simple remote attestation scenario

Measurement systems are the primary concern of this paper and are described in section 2.2 in more detail.

Figure 2.1 illustrates a basic remote attestation scenario. It shows the key components of a remote attestation scenario, the **appraiser**, the **attester**, the **measurer**, and the (target) **application**. The figure also indicates the types of messages passed between these components.

2.2 Measurement Systems

This work is primarily concerned with the measurement system component in the larger context of remote attestation. While measurement systems take many forms, this work is specifically concerned with addressing the unique difficulties and costs of runtime application measurement techniques. These unique difficulties and the current deficiencies in the state of the art measurement systems, which were foreshadowed in chapter 1, will now be described in detail in this section.

In this section, we will first discuss static measurement systems. Then, we will proceed to discuss the focus of this work: dynamic measurement systems.

2.2.1 Static Measurement

To establish trust during remote attestation, the appraisal system needs to identify the remote system. Then, it must request and subsequently verify evidence regarding the remote entity's hardware and software configuration. [6, 18, 38]. The evidence supplied by many remote attestation approaches measures only the static properties of the machine and the running software.

Sailer et al.'s integrity measurement architecture (IMA) was one of the first instantiations of the Trusted Computing Groups' *measured boot* attestation process [49]. Measured boot employs trusted hardware on the target machine, such as a TPM (*trusted platform module*) chip [19], to measure and hash each successive software component, by the preceding software, as it is launched during system boot. Each software hash extends a running hash to create a hash chain that succinctly stores the specific order of the specific software components launched at startup.

In measured boot, each successive component is measured and hashed with a running hash. The very first measurement of this type is performed with a special operation known as SENTER on Intel and SKINIT on AMD. Upon each iteration, the newly computed hash replaces the old running hash. As such, a hash chain is created that evidences definitively a specific set of software components in a particular sequence. The running hash is safely stored in a *trusted platform module* or a *virtual trusted platform module*, TPM and vTPM respectively. The expected system hash is known to the appraisal system in advance and can be compared readily to measured evidence from target in order to establish trust in the initial configuration.

Property-based static analysis approaches have been proposed by others, often as an alternative to the relatively brittle binary measurements of strategies such as measured boot [26, 42]. SWATT [52] and Pioneer [53] employ pure software-based attestation techniques that depend on verification functions that are specially designed such that any tampering attempt noticeably increases their running time. However, both techniques can only attest static code/data. While SWATT can verify the memory contents of an embedded device with a simple CPU, Pioneer can handle complex CPUs to attest a program's binary code. Likewise, there are other works that share the limitation of only being able to perform static or load-time integrity measurement [23, 33].

The static methods described in this section are very effective at describing initial state. Moreover, some of these techniques can be generalized for and applied to various types of software components, regardless of their unique features and even regardless of size. However, they are not sufficient to establish trust nor maintain trust throughout the runtime life of a piece of software.

Static techniques can not guarantee the accuracy of trust inferences in the face of potential errors and malicious attacks that may occur throughout the life of a target application. Measurements have with it a notion of *freshness*. Measurements lose their freshness due to the increasing uncertainty of change with respect to time elapsed since sampling. As such, other measurement functionality is needed to establish trust throughout runtime.

2.2.2 Dynamic Measurement

After trust is established in the initial configuration of a target entity, the target must continue to provide evidence to remain in trustworthy standing with the appraisal system. As discussed in the previous section, static measurement techniques cannot provide this integrity evidence for the dynamic program properties and verify that applications are behaving *as expected* at runtime.

Several works have studied runtime integrity measurement techniques. Flicker utilizes hardware support for late launch to bind and attest the integrity of executed code with its input/output [32]. BIND cryptographically attaches an integrity proof for a program with the output it produces [54]. However, these techniques are not generalized to handle other generic dynamic program state.

PRIMA extends IMA with information flow integrity measurement on SELinux-like systems with a mandatory access control policy [22]. PRIMA prevents high integrity processes from accessing low integrity data, without intervention and alteration. Semantic remote attestation employs a trusted managed-language virtual machine to attest certain properties of the client programs [20] running under its control, including dynamic state at specific program points.

DynIMA combines load-time integrity measurement with dynamic tracking techniques that instrument program code to perform integrity-related runtime checks [12]. DynIMA only supports dynamic checks that are generic to all binaries, requiring no program specific knowledge. Furthermore, no measurement interface is exposed.

Redas provides dynamic RA by measuring certain structural invariants, generated by tools like GCC, and certain global data invariants detected by Daikon [14, 24]. Redas employs operating system modifications and can only perform continuous measurements at certain pre-defined program points, specifically system calls and the malloc family of functions.

The Java Measurement Framework (JMF) provides a measurement framework to sample the dynamic program state as directed by the security policies for applications running within a Java Virtual Machine. Furthermore, Java Measurement Framework defines a policy language to manually write program-specific runtime integrity policies [58].

Several systems measure the Linux kernel runtime integrity. Copilot monitors certain well-known regions of Linux kernel memory using an external PCI card [40]. Another technique dynamically attests the Linux kernel control-flow integrity [41]. Linux kernel integrity monitoring, or LKIM, verifies the consistency of known critical kernel data structures at runtime [28]. These techniques develop custom measurement frameworks that only apply to the Linux kernel.

The cost of designing measurement systems like the aforementioned property-based measurement systems is very high. Unlike simple hashing, these measurement tools target specific software, in this case specific operating systems, and they cannot be used to target other software, not even other operating systems. Furthermore, they cannot even be guaranteed to function meaningfully between separate releases of the same operating system because critical program features, such as the locations and configurations of data structures, may change.

The costs associated with these richer measurement systems are compounded when targeting user-space applications. Runtime measurers must also be tailored to each specific user-space application and maintained over said application's life cycle. While the costs of kernel targeting measurers like LKIM and PIONEER can be amortized across their large code bases, smaller user-space applications do not benefit. As such, manually customizing measurers for user-space applications prohibitively increases development time.

Furthermore, the expected values that describe runtime behaviors are costlier to generate than they are with simple hashes. A hash is generalizable, for all intents and purposes, to any software regardless of features or scale. The *golden value*, or expected outcome, can be computed easily and stored very compactly for future appraisals. Runtime measurements, however, target much richer features that vary significantly between applications and even separate releases of a single application. As such, the expected outcomes also vary significantly.

The burden of generating outcomes for runtime measurement scenarios often falls to manual specification by an expert. Such an expert must have the necessary knowledge and training in two domains. First, they must be well informed in the purpose and implementation of the target application. Second, they must be well informed and trained to identify and adequately evidence critical program features for trust based decision making. Consequently, this manual process makes the generation of expected outcomes and ultimately the prospect of runtime measurement in general very undesirable.

Chapter 3

Measurement System

Measurement systems need to be able to analyze the execution state of arbitrary pieces of software. These measurements systems are very complex and challenging to write. A large portion of the cost of developing measurement solutions for specific applications can be attributed to the challenge of building from scratch the core measurement functionality that is common to the measurement needs of all applications. In this section, we address this common functionality and the lack of general purpose measurement systems with a defined and common evidence querying interface for native applications.

This chapter introduces the MSRR Measurement System. The MSRR measurement system is a novel lightweight measurer that is agnostic of the target application. MSRR provides the core functionality to sample the execution state of a process. This measurement system decouples the application specific policies, which are discussed in chapter 4, such that an expert may focus on writing high-level policies, rather than building an entire measurement system from the ground up for each user-level application.

3.1 MSRR Architecture & Capabilities

The MSRR Measurement System is a prototype measurement system that is application agnostic. The MSRR Measurer is driven by a low-level querying language, the MSRR Evidence Querying Language (MSRR-EQL or EQL), that allows attestation systems to invoke it to gather evidence for would-be appraisals by a remote appraisal system.

While the MSRR Evidence Query Language can be invoked directly, it has been designed to be used with a higher-level measurement policy language, such as the MSRR Policy Language which is described in the chapter 4.

This section describes the measurer architecture in detail, its usage and capabilities, including the details of the MSRR Evidence Query Language.

3.1.1 Measurement System Role in MSRR System

The measurement system, in the context of a remote attestation scenario such as the one described in section 2.1, serves as a slave to the attestation system. The attestation system invokes the measurer, as needed to form evidence for remote appraisals, via the Evidence Querying Language to direct the measurer to attach to specific target applications and to gather evidence. The measurement system interrupts the target application, samples its execution state, and releases it; all as needed to serve the attestation system.

In order to serve measurement, the measurement system requires access to meta-information, specifically application debug symbols, which are generated at compile time at a per-application basis. This debug information provides the measurement system with necessary meta-information to leverage source-level references to code regions and application features in the EQL and resolve them to

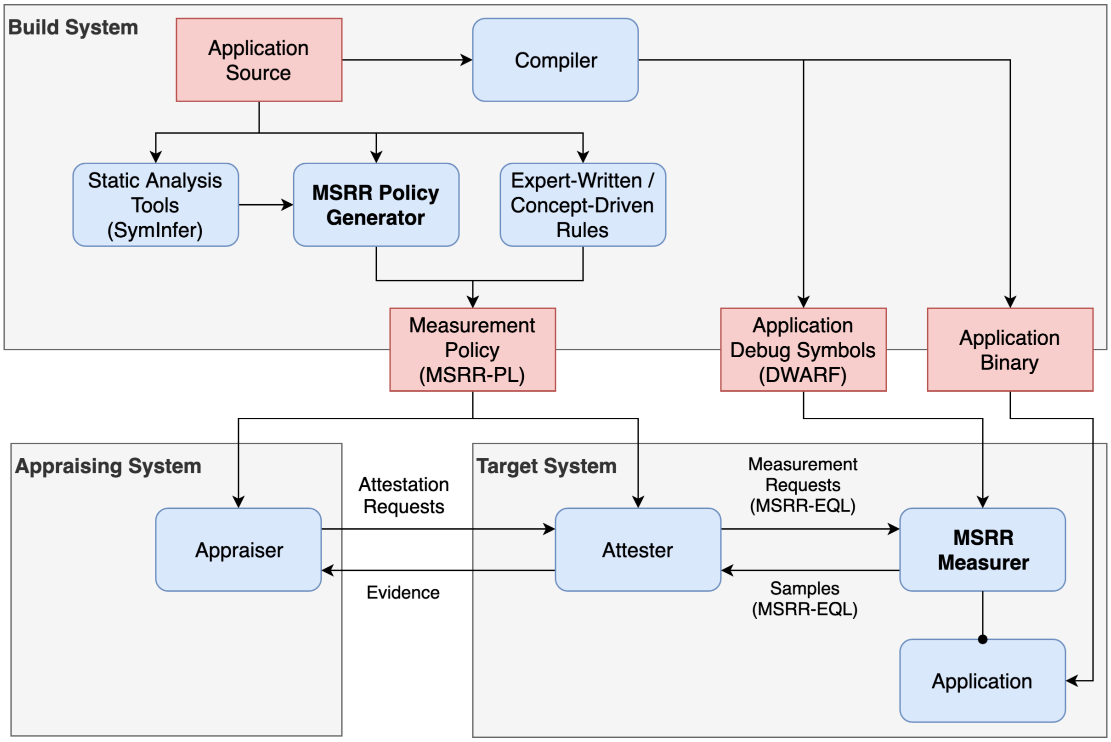


Figure 3.1. Architecture of the MSRR tool suite

their concrete memory addresses during the actual runtime of the target process.

The EQL and the debug information is described in detail in subsection 3.1.5 and subsection subsection 3.1.10, respectively.

Therefore, the MSRR Measurement System, is a piece of software that operates on behalf of the attestation system to sample data from the application. It has access only to the queries provided by the attestation system, the debug information of the target application, and the execution state of the target application.

The measurement system is powerful in that it has the ability to stop, instrument the target's code, read and write to the applications data. As such, these functions have been limited in the followings ways. The measurement system is limited to communicate in response to the attester only, as a remote-procedure-call slave to the attestation system. The measurement system may not alter the outcomes of the target application. However, it is permitted to halt the operation of the target application in order to take measurements. In order to stop and measure the target application at desired code regions, the measurer may temporarily instrument the target application.

That said, a measurer must never change the outcomes of the target application. Any and all changes must be transient. Furthermore, the interruption and slowdown of the target application should be minimized. And, the above impacts of the target are only permissible for the measurement system when serving the attestation evidence queries, and nothing more.

As follows from the slave relationship, the measurement system itself is unaware of the intent of the target application and consequently the expected behavior. The expected behavior is addressed by the measurement policies which influence the measurer only indirectly through the appraiser and attester. The measurement policy language is discussed in chapter 4.

Figure 3.1 illustrates the architecture of the entire MSRR suite. The lower half of the figure contains the remote attestation scenario describe in section 2.1 with the MSRR Measurement System inserted in the place of the generic measurer. The upper portion of the figure demonstrates the build context which illustrates the source of both the application binary and the debug symbols which are used by the target entity. Note that the policy language related nodes refer to separate components of the MSRR Measurement Suite, which are discussed in chapter 4 and chapter 5.

3.1.2 Measurement System Usage & Example

This section provides an example of the usage of the MSRR Measurement System to provide some mental scaffolding for the remaining sections that detail the framework. This example will show how an attester can request measurements of a simple program written in C by sending MSRR-EQL queries to the MSRR measurer.

The example will show each attester-measurer exchange. For each step, we will show the EQL query from the attestation system to the measurement system and the resulting response. We show the EQL query first in a command-line short form, for convenience, and then follow it with the full JSON-RPC invocation. After, the EQL query, we show the result from the measurer, also in both in both the command-line short form and the full JSON-RPC.

The command-line short form is used by the interactive interpreter detailed in subsection 3.1.7.

Example Code The following measurement exchanges will assume the target process is executing the following simple program written in C.

Listing 3.1 Measurer 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);
  }
}
```

Exchange 1 - Attach measurer to target process This exchange shows the attestation system requesting measurement system to launch the target application and initiate communication with it.

Query (launch_as_target ”/path/to/example_binary”)

JSON-RPC

```
{
    "jsonrpc" : "2.0",
    "params" :
    {
        "path" : "/path/to/example\_binary",
        "type" : "launch_as_target_expr"
    },
    "method" : "eval",
    "id" : 1
}
```

Result (void)

JSON-RPC

```
{
    "jsonrpc": "2.0",
    "result": { "type" : "void_result" },
    "id": 1
}
```

As demonstrated here, all exchanges will have a unique *id*, which ensures that the responses are properly mapped back to the corresponding request.

The *void* value returned, for methods that can fail, indicate that the operation was successful. In the event of a failure, an *error* value is returned with failure details. Success, in the case of the *set_target* function indicates that the measurement system has been attached to the target and that it is ready for measurement.

Exchange 2 - Sample the call stack immediately The next exchange uses the MSRR-EQL interface to request an on-demand measurement that samples the target application's call stack.

Query	(measure (call_stack))

JSON-RPC	{
	"jsonrpc": "2.0",
	"params":
	{
	"feature" : { "type" : "call_stack_feature" },
	"type" : "measure_expr"
	},
	"method": "eval",
	"id": 2
	}

Result	(sample (call_graph_value ”main” (call_graph_value ”sleep” (call_graph_value ”__nanosleep_nocancel”))))

JSON-RPC

```
{
    "jsonrpc": "2.0",
    "result":
    {
        "data" : {
            "method_name" : "main",
            "children" :
                [{
                    "method_name": "sleep",
                    "children" :
                    [{
                        "method_name":
                            "__nanosleep_nocancel",
                        "children" : [],
                        "type" : "call_graph_value"
                    }],
                    "type" : "call_graph_value"
                }],
            "type" : "call_graph_value"
        },
        "label" : null,
        "occurrence" : null,
        "type" : "sample_result"
    },
    "id": 2
}
```

On demand-measurements are the simplest types of measurements. The measurer attempts to serve them and reply back to the attestation system immediately, either with the sampled record itself or an error such as a feature out-of-scope error.

In this case, the measurement was successful and the measurement system responded with an instance of the call graph object, which contains the stack frames of the target at the time of measurement.

Exchange 3 - Store a measurement of variable C in function main Our next example has the attestation system requesting the measurement system to register a *monitoring* measurement. This monitoring measurement will sample the value of a local variable, in this case *c*, every time the target application reaches a specific code location.

Command	(hook (reach (method_offset_location ”main.c” ”main” 8) true) (action (store (measure (var ”c”)))))

JSON-RPC

```
{
    "jsonrpc": "2.0",
    "params":
    {
        "event" :
        {
            "location" :
            {
                "file_name" : "main.c",
                "function_name" : "main",
                "offset" : 8,
                "type" : "method_offset_location"
            },
            "repeat" : true,
            "type" : "reach_location_event"
        },
        "action" :
        {
            "expr" :
            {
                "feature" : {
                    "identifier" : "c",
                    "type" : "variable_feature"
                },
                "label" : null,
                "type" : "store_expr"
            },
            "type" : "action_expr"
        },
        "type" : "hook_expr"
    },
    "method": "eval",
    "id": 3
}
```

Response	(void)

JSON-RPC

```
{
    "jsonrpc": "2.0",
    "result": { "type" : "void_result" },
    "id": 3
}
```

Monitoring measurements are created with the use of the EQL *hook* expression. Hooks in MSRR associate some event descriptor (*event*) and an action (*action*). In this case the event is a *reach* function and the action is the store of the measurement of a specific local variable.

A reach event is a trigger that fires every time the target application reaches a specific code location. In this case, we have specified an instruction offset 5 of the main method, which happens to be the following instruction:

```
    printf("c=%d\n",c);
```

The response of this exchange is a void result, which indicates that the hook was registered successfully.

Exchange 4 - Retrieve stored samples. In this exchange, the attestation system requests the measurement system to send all samples that have been taken and stored from previous monitoring measurements. Provided that the location specified in exchange 3 has already been reached, we should get at least one sample of variable *c*.

Query	(retrieve)

JSON-RPC	{
	    "jsonrpc" : "2.0",
	    "params" : { "type" : "retreive_expr" },
	    "method" : "eval",
	    "id" : 4
	}

<table>
<tr>
<td>Result</td>
<td>(sample_set
    (sample (int_value 33))
    (sample (int_value 34)))</td>
</tr>
<tr>
<td>JSON-RPC</td>
<td>

```
{
    "jsonrpc": "2.0",
    "result":
    {
        "samples" : [
            {
                "data" : {
                    "value" : "33",
                    "type" : "int_value"
                },
                "label" : null,
                "occurrence" : null,
                "type" : "sample_result"
            },
            {
                "data" : {
                    "value" : "34",
                    "type" : "int_value"
                },
                "label" : null,
                "occurrence" : null,
                "type" : "sample_result"
            }
        ],
        "type" : "sample_set_result"
    },
    "id": 4
}
```

</td>
</tr>
</table>

The measurement system returns a set of samples from its sample buffer. In this example we retrieved two samples of *c*.

Example Follow-up & Take aways In practice, these low-level queries become quite complicated in order to serve the needs of complex relationships between the many aspects of a target applications. Fortunately, these queries are handled automatically by the attestation system in an effort to form a cohesive proof for an appraisal. This process, in the context of the full MSRR Measurement Suite, is inevitably produced through the high-level MSRR Policy Language measurement policies, which are described in chapter 4. As such, the measurement writer will only need to concern themselves with MSRR-PL, and even then, only when they desire to augment the automatically generated MSRR-PL policies as described in chapter 5.

For more information on the evidence query language, refer to the detailed description in subsection 3.1.5. Furthermore, chapter 7 will provide realistic examples in the context of the full MSRR suite as case studies.

3.1.3 Measurement Types

In this section we present a classification of the measurement types supported by the MSRR Measurement System. The two main categories of supported measurement types are: *on-demand measurements* and *monitoring measurements.*

On-demand measurements are those that are sampled immediately, at the time of request. The measurement system immediately interrupts the target application, samples the requested state of the target application, and then returns the results back to the attestation system.

In the event that a measurement cannot be performed immediately, such as when a target feature is undeclared or out of scope, the measurement fails. Upon such failure, the measurement system responds immediately with an error value.

In practice, on-demand measurements are typically those triggered by an external event to the target application. For example, something on the end of the attestation system or appraisal system could change in such a way that it warrants the need for a fresh sample of some previously measured feature or perhaps a first measurement of a previously ignored feature.

Monitoring measurements are those that can be executed either periodically or upon some trigger, typically a reoccurring one. In either case, these measurements are registered with the measurement system immediately upon receipt of the request. At this time, the measurement system responds simply to indicate success or failure of the registry of such measurement.

Once registered, the measurement system serves the monitoring measurements in accordance with their specification; that is, when their trigger fires. The samples taken in service of monitoring measurements are stored into a buffer for later retrieval, along with meta-information including a timestamp and a label which associates the sample to a specific iteration of the given monitoring measurement.

In practice, event-triggered measurements tend to comprise the bulk of measurement. In accordance with our policy language and the types of remote attestation systems that they inform, a host of monitoring measurements are registered upon attaching to the target application. From that point forward, the appraisal system periodically retrieves the buffered samples to determine validity/trustworthiness of the target application in accordance with said policy. The policy language is described in detail in chapter 4.

3.1.4 Supporting Remote Communications

We expose the MSRR Measurement System's functionality as a JSON-RPC API that the Evidence Query Language communicates over. To this end, the EQL and all of its associated types, expressions, and results are JSON-able.

JSON is a popular format for object serialization that is considered more condensed and easier to read than XML [1, 29]. Considering this and the requirement that the MSRR measurer be invocable by the attester, but not the other way around, the remote procedure call specification over JSON was suitable for MSRR. Moreover, the session-less one-time remote procedure calls approach lends itself well to allowing the measurer to serve multiple attesters, as might be the case in real world remote attestation scenarios.

In the MSRR-EQL, the JSON serializations of each type are implemented in a consistent and sane manner. All types, expressions, results, errors, and otherwise possess a "type" property. The type property defines what MSRR data structure, following a consistent naming pattern.

All datatypes are represented in snake case, and if they inherit from anything, that term is at the end. For example, a hook expression, or as implemented, `HookExpr`, has the type of "hook_expr" in the JSON form. A measurement sample result is implemented with the class name `SampleResult`, and its JSON type is "sample_result." All types in the MSRR suite take this form.

For examples of the EQL, please see the simple step-by-step introduction in subsection 3.1.2, the details EQL description in subsection 3.1.5, or the case studies in chapter 7.

3.1.5 MSRR Evidence Querying Language

The MSRR Measurement System has a robust querying language, MSRR Evidence Query Language (MSRR-EQL), that provides the interface for the attestation systems. The evidence querying language allows the attestation system to specify in detail **what** features of the target application to sample, **how** the measurer should sample them, and **when/where** to make those samples. This section provides a detailed overview of the EQL.

Table 3.3 shows the categories of the EQL commands and lists the selection of functions that were prototyped for this work. The first column in the table lists the command name followed by its arguments in the second column. The third column lists the *type* of the data returned by the command. The last column briefly describes the actions performed by each command.

The MSRR Evidence Querying Language's functionality is classified into the following categories.

Admin and Setup: These commands allow the appraisal system to attach/detach the measurer to the target program and setup the internal state of the measurer.

Measurement: These commands create measurement instances. The `measure` command launches an on-demand measurement, while the `store/retrieve` commands buffer and dump samples from monitoring measurements.

Features: These functions are used to describe various properties of the target application that can be subject to sampling. Features are the subjects of and delivered as parameters to the *measurement* commands.

Snapshots: These commands create and manage snapshots of the target appli-

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			
call_stack	-	feature	Create a *feature* representing the call stack.
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.

Table 3.3. Extensible function interface for the MSRR evidence query language

cation process. Application snapshots are used in a special optimization that MSRR provides which allows potential savings in performance for particularly expensive measurement tasks. This optimization is discussed in subsection 3.1.8.

Events and Hooks: These commands allow for the registration and management of monitoring measurements by constructing events and hooks to perform such measurements.

Locations: These functions are used to specify various code locations for the reach event command.

Control Functions: These commands provide a higher-level interface to create more sophisticated measurements that will only be triggered if and when certain program properties (themselves, determined through earlier measurements) or conditions exist.

Our MSRR implementation allows for the straightforward extension of this command interface.

The MSRR queries utilize values of type `void`, `boolean`, `integer`, `string`, `feature`, `sample`, `sample_set`, `event`, `action`, and `location`.

The `void`, `boolean`, `integer`, and `string` types function as expected.

Measurement functions use the `feature` type to describe a feature in the target application to measure. The `sample` type has two main components, the data component and a type descriptor. The `sample_set` is a set of samples. The data component is the raw data taken in the sample. The type component captures the type of the raw data, which corresponds to the original feature's type in the source language. For example, a measurement may sample an *int* (C integer) of value 5; the resulting *sample* instance would house *5* in the data field and type *integer* in its type field. The `event` type is used to create *event* functions that trigger measurements periodically or when certain conditions are met. The `action` describes any EQL expression as an object to be explicitly executed later, like a parameterless lambda expression; action is currently used by the `hook` function to invoke other MSRR commands when the corresponding event is triggered. *Reach* functions use the `location` type to identify a code region to take samples.

There is a special data type called *error* for error handling. Any function will return an error upon failure. The error type includes with it details relating to the specific failure.

For examples of the EQL in use, please refer to the initial example in subsection 3.1.2 or the additional examples in chapter 7.

3.1.6 Supporting Monitoring Measurements

Monitoring measurements are those that can be registered in advance for later sampling. These can be a delayed one-time measurement or a recurring measurement. They can be triggered by timers or upon the target reaching critical code regions.

In order to support these measurements, the MSRR measurer requires a few capabilities. It needs to be able to store measurements in a buffer for later retrieval. It must expose, in the EQL, a method to retrieve stored measurements. The measurer must also maintain each active monitoring measurement, as hooks, in a registry for later firing, and to allow them to be disabled or removed.

The monitoring measurement samples are stored in a sample buffer along with meta-information to await eventual retrieval by the attestation system. The meta-information includes a timestamp of the measurement, a reference to the hook expression that engendered it, and a sample identifier unique to that monitoring measurement. Upon retrieval, the sample is sent with the metadata so that the attestation system can determine which samples belong to which iterations of the monitoring measurement, in order to apply them together as a coherent set of parameters to relevant rules. Rules and rule applications are described as part of measurement policies in chapter 4.

Measurements are registered by the *hook* command. The hook command expects a type of *event* and a type of *action*, which is simply any expression in the EQL language passed as an object to be evaluated explicitly later.

An event can be either a timer event or one of many location reach related events.

For timer events, all timer hooks are registered into the active hook set in the MSRR Measurement System context with their corresponding action expression. Upon expiry of the timer, the action's corresponding expression is evaluated. Then, non-recurring timers are evicted from the active set.

For location-reach events, the hook is registered as well into a set of active hooks. Additionally, the low-level backend is invoked to set up the proper controls for the target application. This process is detailed further in the implementation subsection 3.2.3. In the registry, we store a reference to any information needed to relate the hook entry with the backed controls such that when one is disabled or removed, both shall be.

In some cases, more complicated events can be registered. Consider a case where one wishes to take a measurement at location `L1`, but only at the first occurrence of `L1` immediately after each occurrence of another location `L2`. Examples like these can be performed using the control logic provided by the EQL; specifically, by nesting the hooks and sequencing them with commands that subsequently disable the hooks.

In kind with many real world remote attestation policies, these types of MSRR-

EQL queries can get very complicated. Fortunately, when the whole MSRR suite is leveraged, the measurer developers only needs to focus manual effort at the high level of the MSRR Policy Language.

3.1.7 Evidence Querying Language Interactive Client

Along with the MSRR Measurement System, we provide the MSRR Evidence Querying Language Interactive Client. The EQL Interactive Client is a tool that allows one to connect to an MSRR Measurement System from a command line terminal and execute EQL queries directly.

This tool has been useful for debugging and extending the MSRR Measurement System itself. We believe that it may also prove useful for scenarios that this work does not consider, such as those outside of remote attestation where the high-level driving of the MSRR-PL is not required, and yet the sampling of various application features is a requirement.

3.1.8 Snapshot & Release Optimization

Measurements in the MSRR Measurement System are, by default, considered *direct measurements.* Direct measurements are those that stall the target piece of software for the duration of sampling. In some cases, measurements can be very large or complex such that they impose a significant overhead and slowdown of the target application processes. For such large measurements, we have implemented an optimization strategy that utilizes a unique type of measurement called *snapshot measurements.*

The following is the typical workflow for a direct measurement. Upon receiving a measurement request, depending on its type, MSRR either performs an immediate sampling or schedules a sampling event. To ensure coherency, the actual measurement interrupts and stalls the target application for the duration of the querying and processing of all data requested. In other words, data relevant for a specific expected behavioral rule (as seen in chapter 4) applications must be taken from the same sampling point.

Typical Measurement Workflow:

1. The measurer receives a measurement request from the attestation system.
2. The measurer interrupts the target application.
3. The measurer searches for and collects the desired data, if available.
4. The measurer releases the target program.
5. The measurer packages the data and sends the results to the attestation system.

The snapshot measurement strategy is reserved for heavy-weight measurements. This strategy takes a full snapshot of the target application and then releases it. The actual measurement query is then performed on the snapshot itself. As such, for heavy measurements or composite measurements, the naive full snapshot actually improves performance.

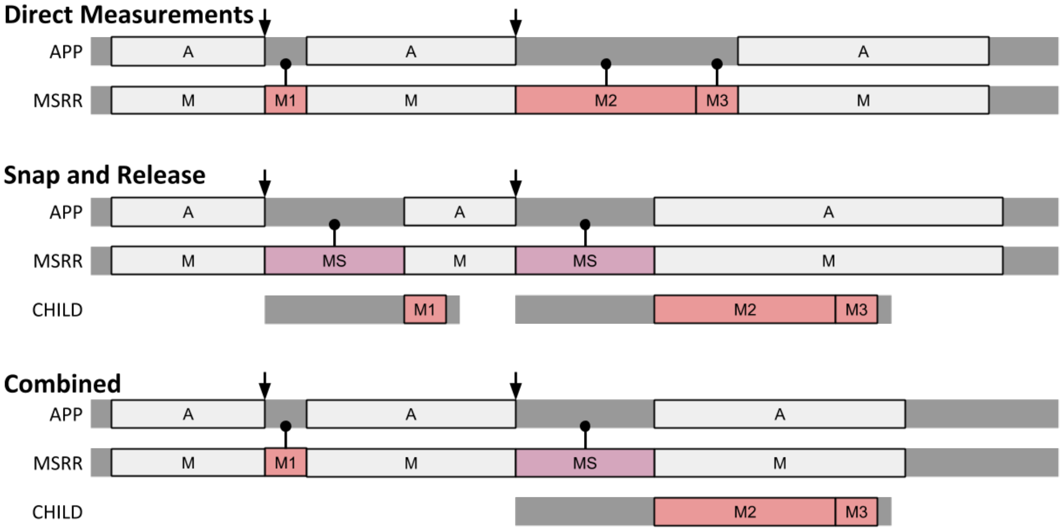


Figure 3.2. Measurement execution timeline comparison for direct measurement and snapshot measurement strategies

Snapshot and Release Workflow:

1. The measurer receives a measurement request from the attestation system.
2. The measurer interrupts the target application.
3. The measurer takes a snapshot of the target application's state.
4. The measurer releases the target program.
5. The measurer collects the relevant data from the snapshot and decommissions it.
6. The measurer packages the data and sends the results back to the attestation system.

Figure 3.3 compares the direct and snapshot measurement strategies for several scenarios. The snapshot measurement strategy is provided to lower the measurement overhead when direct measurement is likely to interrupt the target program for longer than the cost of taking the snapshot. In Figure 3.3, the measurement system receives the following measurement requests: first, a measurement request for **M1** and later a measurement request for the simultaneous measurements {**M2**, **M3**}. The first case uses direct sampling exclusively. The second example employs snapshot sampling. However, the snapshot measurement for M1 produces no performance benefit but snapshot for {M2, M3} does. The last case demonstrates the ideal use-case, where direct measurement is used for M1 and snapshot measurement is taken for the {M2, M3} request. Snapshot measurements, currently, need to be explicitly requested through the EQL snapshot commands.

3.1.9 Necessities for Performing Measurement

In order to perform measurement, the MSRR Measurement System must have access to the execution state of the target process. The measurer must also be privileged to be able read the memory of the target process. Additionally, the measurer must be able to halt and resume the target process to perform measurement, in order to ensure atomicity of related sets of measurements.

While arbitrary immediate interrupts can be performed relatively easily, the measurement system must also be able to halt the target application at specific locations. To perform this task, the measurement system must be authorized and able to temporarily instrument the target application. This is discussed further in the implementation section.

As such, we have designed the measurement system to handle these capabilities. Furthermore, we assume that the measurement system process will have the necessary level of privilege on the host machine to carry out its duties.

3.1.10 Necessities for Informing Measurement

The ability to access execution state alone is not sufficient to serve measurement requests, if the MSRR Measurement System does not have the information needed to make sense of the target state. To facilitate this, the MSRR Measurement System assumes that meta-information is present, either residing with the measurer or the target binary, which details critical pieces of information about the target application's code, identifiers, types, variables, etc., in order to serve MSRR-EQL queries.

Our MSRR Measurement System gets this information in the from of the debug information that modern compilers have been generating and perfecting as debuggers have evolved. These debug symbols are easily produced by such a compiler and can be packaged with the target binary or even be stripped away and made available to MSRR Measurer via other means.

These debug symbols allow the EQL to express measurements at the high source-level identifiers and code regions. Then, the measurer resolves these source-level references down to their concrete memory locations at runtime by using the debugging information.

The details of the debugging information is described in subsection 3.2.4.

3.1.11 Security/Verifiability Implications

A concern in the realm of remote attestation and measurement is the problem of establishing trust in the measurement system and the measurement samples it produces. Such verification is often approached by a technique known as *meta-measurement*, in which a separate meta-measurement system measures the application measurers themselves.

However, the techniques employed by MSRR focuses on reducing the manual effort of building tailored measurement systems from the ground up. To this end, this work does not explore meta-measurement and considers it beyond the scope.

3.2 Implementation

We have implemented the MSRR Measurement System in layers. The EQL interface along with the general measurement context resides in the top layer which sits on top of a modified, faceless, version of the GNU Project Debugger, GDB [15, 16], which peers into the target application. In between these two layers, resides a middle layer that associates the two in such a way that, should the backend need to change, the top layer need not be modified.

This sections describes in detail the MSRR Measurement context, the middle layer, and the modifications made to GDB.

3.2.1 Measurer Layers

The MSRR Measurement System's implementation is broken up into three separate layers. Due to the fact that the measurement system leverages an existing system, the GNU Project Debugger (GDB), to instrument and read the memory of the target application, we have separated the main measurer context from the GDB backend. As such, the MSRR context forms the top layer and GDB forms the bottom layer. An adapter layer sits between the two to relate the general measurement context with the symbolic debugger backend.

Figure 3.3 illustrates the MSRR Measurement System in the context of a

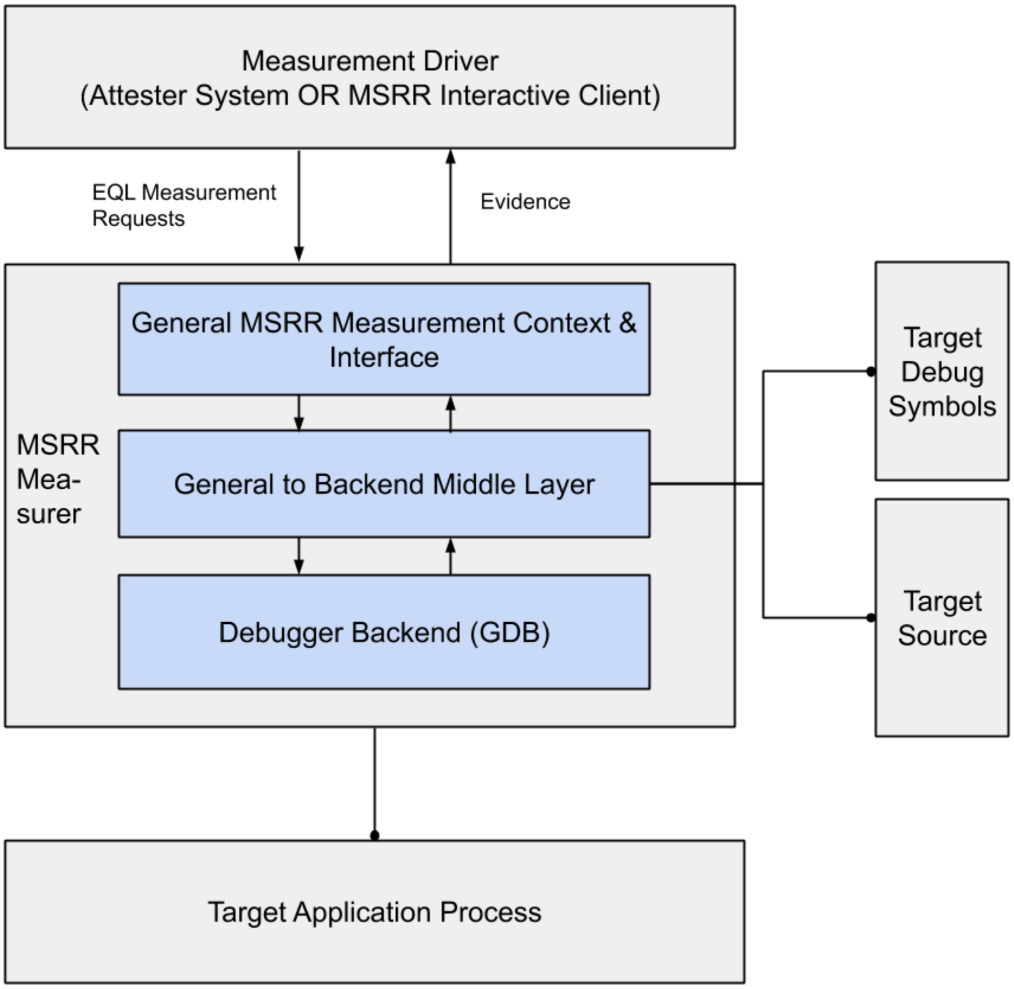

Measurement Driver
(Attester System OR MSRR Interactive Client)
EQL Measurement Requests
Evidence
MSRR Measurer
General MSRR Measurement Context & Interface
General to Backend Middle Layer
Debugger Backend (GDB)
Target Debug Symbols
Target Source
Target Application Process

Figure 3.3. Architecture of the MSRR measurement system

simple attestation scenario.

Upper Layer: Measurement Context & Interface The top layer contains the general measurement context; that is, everything independent of the specific backend. As such, the measurement context layer contains the MSRR Evidence Query Language interpreter. It also contains most of the measurement state, including the measurement buffers and the monitoring measurement hook-event registries.

Lower Layer: Debugger Backend This layer is the component that is in charge of controlling the target application, reading its memory, and utilizing the debug info. In short, it is the layer that does all interfacing with the target application.

We have prototyped the MSRR Measurer using the GNU Project Debugger, GDB [15, 16]. GDB provides the functionality to control the execution of the target application, to read its memory, and to interpret the DWARF debug symbols.

In an effort to make the MSRR future-proof, we have abstracted the GDB backend into its own layer and have setup a middle layer to minimize modifications made to the backend/GDB and the measurer context in the event should such a backend change.

That said, GDB has been modified slightly to be able to function in our system. Specifically, its own frontend and top level control structure has been stripped away. As such, the middle layer interfaces with the many functions in GDB's backend that inevitably, in its original case, serves their own command structure.

Middle Layer: Adaptor The middle layer serves as an adapter for the Measurement Context Layer and the debugger backend. As such, it implements a specific interface; that is, the set of commands the top layer needs to function. The middle layer in turn invokes the proper backend functions and manages any state that the backend might need. For example, for each hook registered in the measurement context, the middle layer stores the associated backend/GDB breakpoint identifiers to properly handle the setting and deletion of such.

3.2.2 GDB Considerations

We have implemented MSRR as an extension of the GNU Project Debugger, GDB [15, 16]. Consequently, our current implementation benefits from GDB's extensive capabilities and infrastructure, but also inherits GDB's limitations.

GDB utilizes the DWARF debug symbols to drive its measurement capabilities [13]. As such, MSRR Measurement System requires its meta-information about the target application, as discussed in subsection 3.1.1, to be in the DWARF format. The DWARF data enables the measurement system to easily find various program features using source code level descriptors, and simplifies policy generation, either manually by an expert or by other automated means, using the descriptors found in the code source. The DWARF debug information is detailed further in subsection 3.2.4.

Our GDB-based MSRR measurement system utilizes hardware features, OS interfaces, and program instrumentation to provide its measurement capabilities. For instance, the EQL's code region *reach event* functions employ GDB's *breakpoint* functionality. GDB implements breakpoints using either the built-in hardware breakpoints, if available, or as software breakpoints using program instrumentation to replace a program instruction with a trap. Likewise, the MSRR Measurer utilizes GDB's *syscall* feature to pause the program at system calls for measurements which target interfaces exposed by the OS.

The overhead of code instrumentation in the measurement system is limited because the instrumentation is only inserted during the measurement period, and does not need to permanently slow down the entire program execution.

3.2.3 Tracking Monitoring Measurements

The monitoring measurements, those that are registered and executed later upon some event, are tracked via registries that exist in the measurement context layer. The registries are sets of hook objects that can be either active or disabled.

Hook objects are essentially pair associations of some event to some EQL action. Timed events are fired when their timers go off and, if non-recurring, are removed from the active set. Recurring timed events are reset after each firing.

Upon the firing of a timed event, the measurer context layer sends a request to interrupt the target application immediately. The GDB backed serves this request by sending an interrupt signal to the target application. The EQL action expression is interpreted in the measurer context and the backend is invoked accordingly to sample any measurements therein. Finally, once measurement is complete, the target is sent a signal to continue.

Location reach events are implemented with a hook entry in the registry as well. When the hook is registered, the GDB backend is invoked to instrument the target application with a trap just before the target location, which in the context of GDB is called a breakpoint. Each breakpoint gets a unique identifier which MSRR stores a copy of in the middle layer to maintain an association between an MSRR hook and the corresponding GDB breakpoint.

The middle layer then listens for an event from GDB that would signal a breakpoint has been reached. Upon the reaching of a breakpoint, the GDB identifier is translated back to the originating hook and the corresponding EQL action is executed.

3.2.4 DWARF Debug Symbols

The MSRR Measurement System is prototyped with the GDB backend and, as such, utilizes the DWARF (Debugging With Attributed Record Format) debug symbols to drive its measurement capabilities [13]. The DWARF data format records the necessary information to empower the measurement system to find various program features using source code level descriptors. This greatly simplifies both the evidence query language interface and the policy language, described in chapter 4, that is build upon it.

DWARF debug symbols can be produced optionally by most state of the art compilers, for example by using the '-g' option with GCC. We assume that the DWARF debug symbols are available to the measurer and the appraiser, even if they were stripped from the target binary program.

The DWARF format uses Debugging Information Entries (DIE) to describe various features of a target application. Each DIE is comprised of a tag, to identify the feature, and a series of attributes which describe the feature. Each attribute is key-value pair.

Listing 3.6 DWARF format example: target C program

```
int a;
void foo()
{
    register int b;
    int c;
}
```

Listing 3.7 DWARF format example: debug information entries

```
<1>:    DW_TAG_subprogram
            DW_AT_name = foo
<2>:    DW_TAG_variable
            DW_AT_name = b
```

```
                DW_AT_type = <4>
                DW_AT_location = (DW_OP_reg0)
<3>:        DW_TAG_variable
                DW_AT_name = c
                DW_AT_type = <4>
                DW_AT_location = (DW_OP_fbreg: -12)
<4>:        DW_TAG_base_type
                DW_AT_name = int
                DW_AT_byte_size = 4
                DW_AT_encoding = signed
<5>:        DW_TAG_variable
                DW_AT_name = a
                DW_AT_type = <4>
                DW_AT_external = 1
                DW_AT_location = (DW_OP_addr: 0)
```

Listing 3.7 shows the DWARF debug information entries that describe the variables from the program shown in Listing 3.6. Keep in mind that debug information entries are used to describe other aspects as well; such entries have been omitted from the listing.

In the figure, we can see the sample C program with a few source level identifiers: three integer variables with ids `a`, `b`, & `c`; and a function with identifier `foo`. The DIEs are `DW_TAG_variable` and `DW_TAG_subprogram` for the variables and the function, respectively. Under each DIE, we can see the nested attributes listed. For example, each of them list the source identifier with the attribute `DW_AT_NAME`. In addition to the identifier, the type information, and the location is also shown for the variable related DIEs.

3.2.5 Performing the Snapshot Measurement

As described in section subsection 3.1.8, we employ an optimization strategy for large and composite measurements called the snapshot and release strategy. The snapshot and release strategy allows one to run expensive measurements in parallel with the execution of the target application, after a full memory snapshot is taken.

To implement this we have utilized the Linux *fork* system call to construct a new *child* process that retains a full snapshot of the original program's processor and data state [45]. The actual measurement happens on the child snapshot. The original target application is allowed to continue after the *fork* command returns.

In order to fork the target application, we leverage the LD_PRELOAD functionality in linux. The LD_PRELOAD option exploits the linux dynamic linker to allow one to bind symbols before any other dynamic libraries load [8]. In MSRR, we use this functionality to hijack the entry method and add a signal handler for a custom signal to the target application. This signal handler forks the target application. After the fork, the original process is allowed to return to its normal execution. The child process waits indefinitely, allowing its memory to be read at will by an MSRR measurer.

At the same time, the MSRR Measurement System spawns a new MSRR Measurer from itself as a child. The original measurer acts as the child measurer's attester, and in this way it serves it the nested measurement intended for the snapshot of the target.

Upon completion of the snapshot related measurements, both the child measurer and the child application process snapshot are disposed.

An advantage of implementing the snapshot measurement via a call to fork is that the target application's memory does not need to be exhaustively copied, at least not in realistic cases. That is, upon a fork, all of the applications memory is marked for copy-on-write [45] and, as such, only the data that the original application process modifies between snapshot time and the end of measurement is actually copied.

3.2.6 EQL Interactive Client

The Evidence Querying Language Interactive Client is implemented using a human readable Scheme-like language to allow the user to easily write simple test queries. The interactive client makes requests via the same method that a real attestation system would and, as far as the Measurer is concerned, is treated in kind.

Listing 3.8 Example usage of the MSRR interactive client

```
(set_target 1001)
>> (void)

(measure (var "c"))
>> (sample (int_value 7))


(measure (var "c"))
>> (sample (int_value 11))

(shutdown)
>> (void)
```

Above is a short use case of the interactive client. In this example, the client sends a query to the measurer to attach to some application which is incrementing, iteratively, some local variable C. The client is then used to sample the variable C with direct measurements. Finally, the client sends the command to terminate the MSRR Measurement System.

Chapter 4

Measurement Policy Language

Measurement policies describe how a measurer should be used to properly and efficiently sample critical program features of a target application. Even with a good application measurement system, such as the one this work proposes in chapter 3, much effort must be expended in understanding the target application and in specifying an attestation system to properly drive the measurer to sample accordingly.

The MSRR Measurement System exposes a powerful low-level querying language that is not intended to be invoked directly when writing attestation systems or their corresponding measurement policies. For this purpose, a higher level language, a *policy language*, that can be compiled or interpreted into the given low-level Evidence Querying Language is essential to making measurement strategies feasible for remote attestation.

This chapter presents the MSRR Measurement Policy Language, or MSRR-PL. The MSRR Measurement Policy Language is a high-level policy language that allows a measurement system writer save much time and effort by describing complicated policies in simple terms. Furthermore, MSRR-PL makes the process of writing measurement systems much mores structured and disciplined, making measurer behavior more efficient, predictable, and finally testable.

In essence, MSRR-PL provides a high-level interface to the MSRR evidence querying language, MSRR-EQL. This allows one to effectively describe complex measurements, their relationships, and the conditions for successful validations of the eventual samples.

We have implemented the MSRR Policy Language as a domain specific language in C++. This chapter is divided into two sections. The first sectiont describes the language, its usage, and its functionality. The second section details key components of the implementation.

4.1 Architecture and Capabilities

The MSRR Policy Language is a prototype policy language designed to describe measurement policies at a high level. The MSRR Policy Language has been implemented with a backend that produces the low-level MSRR Evidence Query Language instructions for the MSRR Measurement System. However, with the proper adapters in place, this policy language can be used to describe policies for future measurers and their own evidence querying languages.

In this section we describe the language itself, via example, and then we continue on to discuss many of the features that the policy language provides.

4.1.1 Components of a Measurement Policy

A measurement policy serves two purposes. The first is to describe some critical subset of the expected behavior of a target application. The second is to describe how the measurement system should be instructed to evidence the

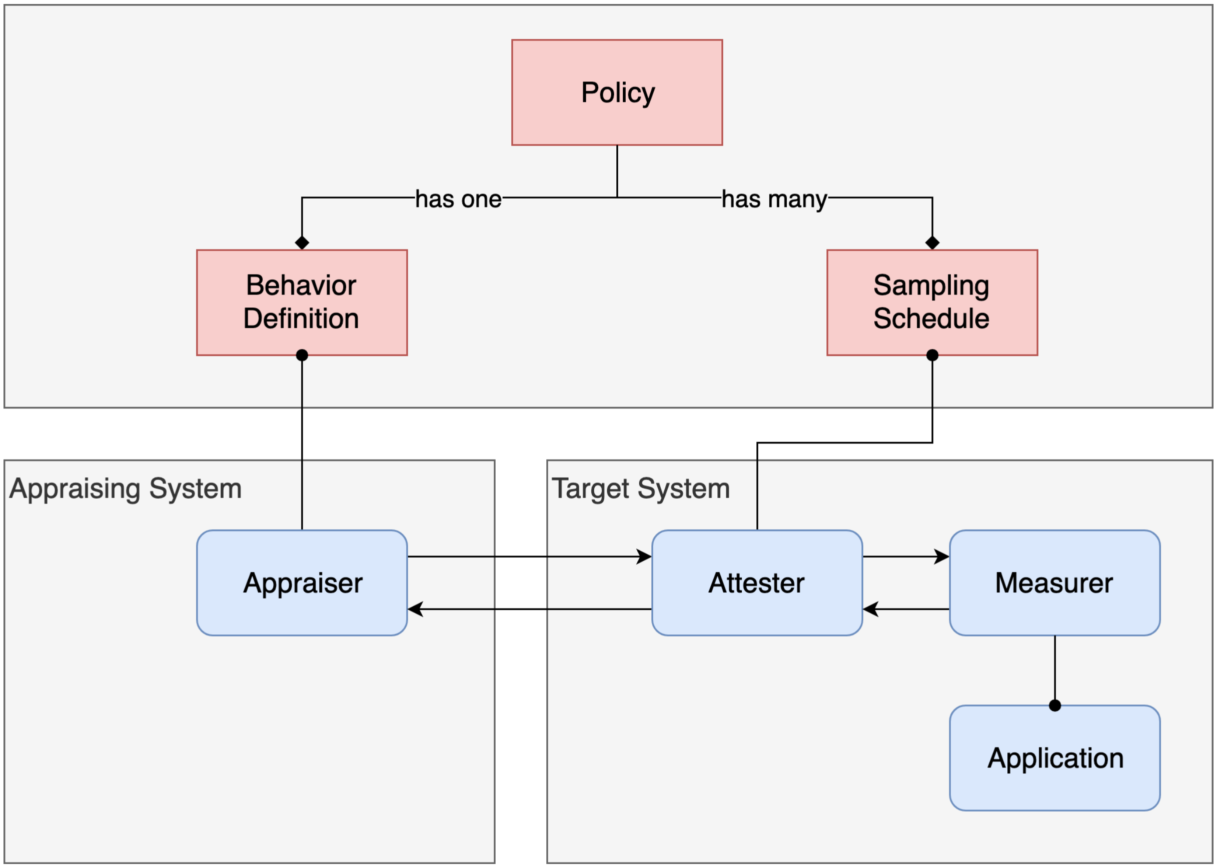


Figure 4.1. Roles of the expected behavior definition and the sampling schedule in the context of a remote attestation scenario

expected behavior.

As such, a measurement policy in the MSRR policy language is broken up into two somewhat overlapping parts: the *Expected Behavior Definition* and the *Sampling Schedule*.

Figure 4.1 shows how the two major components of a measurement policy are used by the components of a typical remote attestation scenario.

Expected Behavior Definition The expected behavior definition describes some subset of the expected behavior of a target application. This component of the measurement policy is measurement system independent, in the sense that it expresses facts about a program's behavior generally speaking. At runtime for a remote attestation system, the appraiser has access to the expected behavior definition of the target application; the measurement system however does not use it directly.

The expected behavior definition is comprised of *rules*. Rules can vary in scope, but they are designed to capture a specific property of a target application. In spoken language, a rule might read something like *"For all instructions, local variable X must be greater than Y"* or *"At instruction I1, the local variable password must equal 'password123' while local variable logged_in equals 'true'."*

Sampling Schedule The sampling schedule describes how the measurement system should be instructed to properly evidence the associated expected behavior definition. The sampling schedule essentially determines how often specific rules should be sampled, under what conditions, or if they should be sampled at all.

For a single expected behavior definition, there may be multiple schedules. However, a single measurement system can be subject to only one schedule per expected behavior definition at a time. For example, a single expected behavior definition may have two schedules: one that samples for each rule periodically at a relatively moderate frequency and another that only measures one of the rules, yet does so very frequently.

Sampling schedules are comprised of sampling directives for each expected behavior definition rule and said rule's parameters. These are directives that express the frequency that each rule is measured and dictate concrete locations of sampling for each rule parameter.

4.1.2 Basic Usage and Example

This section provides an example of how one might use the MSRR-PL to write a simple measurement policy for a simple program written in C. We will show each section of the policy written step-by-step with explanations of the language features that are used along the way.

Example Code The measurement policy that we will write will describe a simple property of the following C application.

Listing 4.1 MSRR-PL example: simple target application in C

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

int main() {
  int x = 2;

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

}
```

Observe that in the C program above, the variable x is incremented by two, each iteration of the while loop, and therefore should always be even. So, we will write a simple policy in MSRR-PL to encapsulate that (A) the variable x should always be even and (B) that the measurer should take periodic samples in order to evidence that fact.

The remainder of this section will, step by step, build the full policy for this scenario. First, we will define the expected behavior and then we will define a sampling schedule.

Describing Properties - Validation Functions First we need to define the property that must be true. Id est, we must describe the evenness of local variable *x*. We do this in MSRR-PL with a *validation function*.

A validation function is essentially any n-ary function that evaluates to a boolean. Generally, such a function for evenness could be expressed as follows:

Listing 4.2 Definition of evenness written in C

```c
bool is_even( int x ) {

    return x % 2 == 0;

} );
```

The MSRR Policy Language differs from this standard C function, but the basic pattern is the same. In MSRR-PL, all validation functions expect a C++ lambda of type *SampleSet* to type *bool*.

The SampleSet is an MSRR collection that houses samples, which are of type *Sample*, taken by the MSRR Measurement System. While the Sample object contains meta-data useful in other contexts of measurement, we are only concerned with two attributes for the time being.

The specific sample can be accessed with a custom label: “x_parameter” was chosen below for the measurement of local variable X. In this case, we are only concerned with the *value* property of the Sample, which is assumed to be of MSRR type IntValue.

As such, we can initialize a new policy and then add a new validation function to it as follows:

Listing 4.3 MSRR-PL example: evenness 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;
            }
        );
```

Describing the Subjects of a Property - *Features* Validations function expect a sampling of some *feature* of the target application as an input. We must now use the Feature datatype of MSRR-PL to declare the feature, local variable X, that will eventually be sampled and passed to the validation function when the system goes live.

This can be done as follows:

Listing 4.4 MSRR-PL example: variable feature X

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

Describing the Location Scopes of the Property - *Locations* The validation function is useful for describing a true-false property about an application, but in most cases these such properties must be scoped to some code region of the target application.

We do this with the MSRR-PL location scope datatypes. Location scopes can take many forms. In this case, we will use the *FileRangeLocation* datatype to capture the body of the loop in the example program. The FileRangeLocation describes a specific range of instructions, by line numbers, within a specific file.

Assuming the file is named “main.c”, the location can be described as follows:

Listing 4.5 MSRR-PL example: loop body location scope

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

Describing the Occurrence Scopes of the Property - *Occurrence Scopes*

So far, we have described a simple property for evenness. We have described the feature that should be even. We have used location scopes to describe where it should be even. Now, we must describe when it should be even.

Occurrence scopes let us specify when a specific code location is relevant for sampling. In this example, the answer is trivial: *anytime* that we are within the loop body.

To encapsulate 'anytime' for a given location, we will use the Origin Occurrence type. An origin occurrence is an unbounded occurrence scope that is used by other scopes as a point of origin/reference. The other occurrence scopes and how they are used with each other is described in subsection 4.1.6.

As such, we describe the occurrence scope as follows:

Listing 4.6 MSRR-PL example: loop body occurrence scope

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

Associating a property with a what, when, and where - *Rules* Now we need to associate the feature and its scopes to the validation function to form a measurement policy rule.

To do this, we will first construct a scoped parameter with the Parameter type. A scoped parameter expects a feature definition and an occurrence scope, which contains within it a reference to its location scope. Then, we will define a new rule with both the validation function and its single parameter.

This is done as follows:

Listing 4.7 MSRR-PL example: scoped parameter and rule

```
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"});
```

Describing the Sampling Schedule As of the last step, we have successfully described the expected behavior of the application with our single rule. Now, we must specify the schedule for which the measurement system will actually take samples to evidence said expectations. This is accomplished using the Sampling Schedule datatype.

Sampling schedules can vary greatly, resulting in a spectrum of configurations with an inversely proportionate relationship between performance impact and completeness of the measurement sample profile. For a single rule, one schedule may be very intense and demand frequent sampling. Another schedule, for the same rule may skip many sampling opportunities, thus decreasing overhead while increasing the risk of missing potential bad states.

For this example, we will construct a schedule to take a single measurement every other time that the execution enters the relevant code regions of the rule. Furthermore, we will set it up so that the actual instruction at which the sample is taken each iteration is random within the loop body.

To do so, we must first create a new sampling schedule which we will call ‘default_schedule.’ Then, we will add a rule schedule to the sampling schedule which will determine how the ‘is_even_rule’ is sampled. A rule schedule is merely the portion of a sampling schedule that covers a single rule. A full sampling schedule is nothing more than a set of rule schedules.

Rule schedules are defined with a sample rate specifier and, for each parameter, a specific sample point. The sample rate specifier allows one to control how frequently a rule is actually sampled. The sample point allows one to control where in a parameter’s associated location scope to actually take the sample.

The schedule for this example will look as follows:

Listing 4.8 MSRR-PL example: sampling schedule and rule schedule

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

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

Full Policy Example We now have constructed a complete policy. This policy has a single rule which describes the evenness property for a local variable in the main iterative loop of the target program. We have defined a single sampling schedule which takes samples at a random location within the loop body, every other iteration of the loop. The full example is displayed in Listing 4.9.

Listing 4.9 MSRR-PL example: full policy

```
Policy policy;

// Build Behavior Definition

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["x_feature"] =
  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"});

// Build Sampling Schedule

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

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

4.1.3 Policy Language Functions & Definition

This section gives an overview of the MSRR Policy Language definition. In the following sections, the specific categories of MSRR-PL features are described

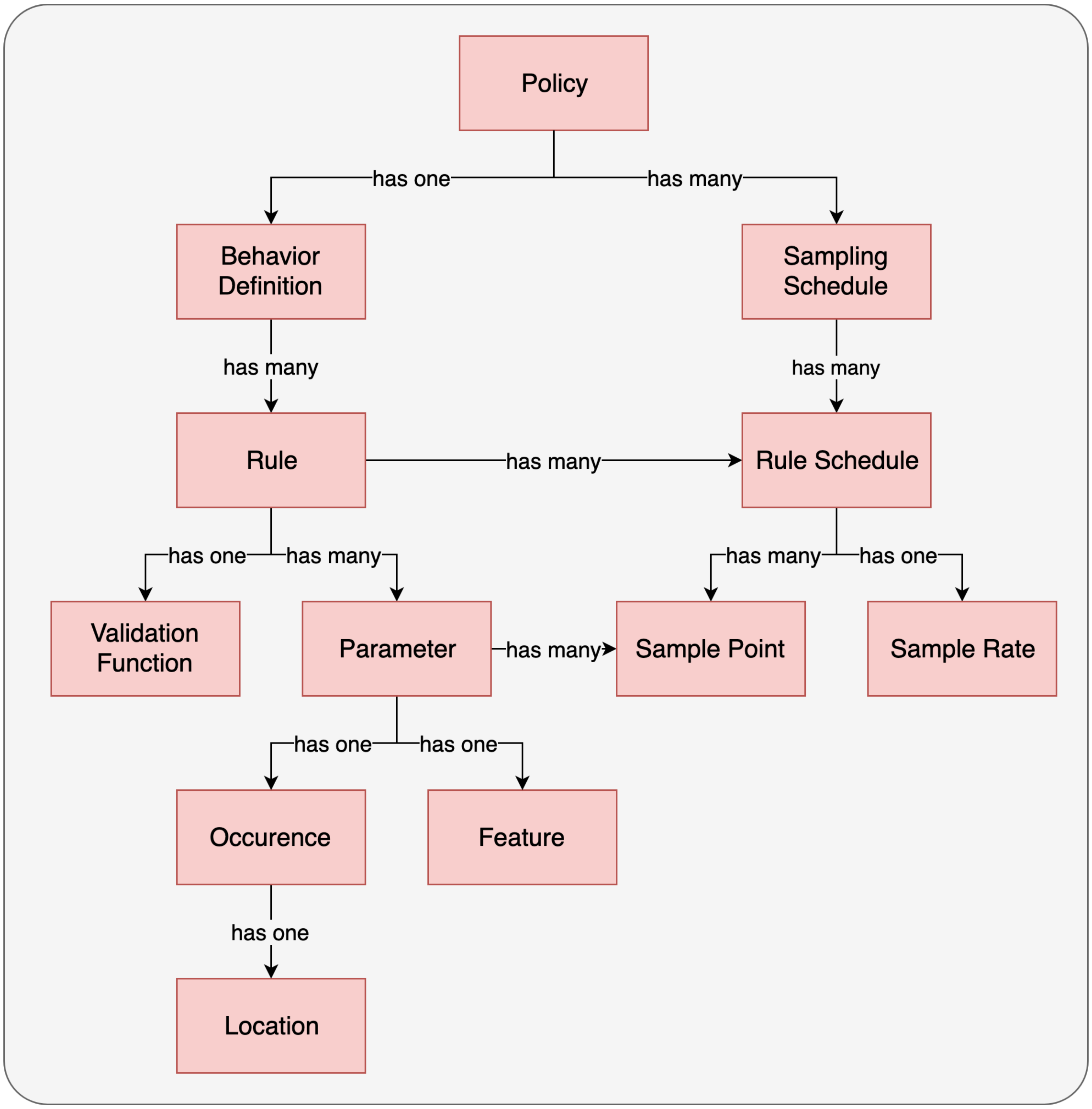


Figure 4.2. Language feature association diagram for the MSRR policy language

in detail.

The MSRR Policy Language is comprised of two major components, the Expected Behavior Definition and the Sampling Schedule. The types of MSRR-PL are detailed below in BNF-like form and highlighted in Figure 4.2.

Listing 4.10 Grammar definition for the MSRR policy language

```
Policy := ExpectedBehaviorDefinition SamplingSchedule*

ExpectedBehaviorDefinition := Rule*

Rule := ValidationFunction Parameter*

ValidationFunction :=  [](SampleSet)->boolean

Parameter := Occurrence Feature

Occurrence := OriginOcccurrence Location
    | NextOccurrence Location Occurrence
    | KthNextOccurrence Location Occurrence
    | FirstOccurrence Location

Location := FileLineLocation string integer
    | FileRangeLocation string integer integer
    | FileMethodLocation string string
    | FileClassLocation string string
    | UnionLocation Location Location
    | IntersectionLocation Location Location
    | DifferenceLocation Location Location
    | SymmetricDifferenceLocation Location Location

Feature := CallStackFeature
    | VariableFeature string
    | RegisterFeature string
    | MemoryFeature string string

SamplingSchedule := RuleSchedule*

RuleSchedule := SampleRate SamplePoint*

SampleRate := EveryIteration
    | EveryOtherIteration
    | EveryKthIteration integer
    | EveryIterationAfterDelay integer
    | ChanceOfSampling integer
    | SkipSampling
```

```
SamplePoint := FileLineSamplePoint string integer
    | FirstLineSamplePoint
    | KthLineSamplePoint integer
    | LastLineSamplePoint
    | RandomLineSamplePoint
    | MethodEntrySamplePoint string string
    | MethodExitSamplePoint string string
```

4.1.4 Validation Functions

The validation function captures the propositional logic that describes the expected states and relationships of said states for some set of features of the target application. The validation function type of MSRR Policy Language, at its core, contains a lambda function of any set of inputs to boolean. The inputs are the measurements taken from various features in the target application, and the boolean result expresses wether said features were as expected or 'good' with respect to each other.

The validation function is exclusively part of the expected behavior definition. That is, it is used by appraisal system to verify that the samples meet the criteria of the policy.

4.1.5 Location Scopes

Location scopes are used to restrict a parameter of a rule to some code region. There are several types of scopes, listed below, which allow one to select some set of instructions via source level features such as the name of a class.

The MSRR Policy Language also provides various set operation types which allow locations to be used together to select more complicated code region sets, such as those with noncontiguous instructions.

The following location types have been implemented to serve the needs of our prototype measurement suite:

- **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.
- **Location 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.

4.1.6 Occurrence Scopes

Occurrence scopes are used with a location scope to bound a parameter to a relative time at, or occurrence of, a specified location.

Occurrence scopes are relative specifiers that are meaningful with respect to each other. The default occurrence scope is the `OriginOccurrence` scope. An origin occurrence can be considered a point of reference to which other occurrence scopes can be related. In other words, the origin occurrence scope itself is unbounded, time-wise, and other occurrences are bound to it.

The `NextOccurrence` occurrence scope is used to describe the next occurrence of a location, following some other occurrence.

For example, consider a scenario where one wishes to sample and compare iterations of some iterator `i` in a loop, at location `first_loop_instruction`. To do so, the OriginOccurrence of `first_loop_instruction` can be used to sample the initial value of `i`. To sample the successor value of `i`, the NextOccurrence scope can be defined at `first_loop_instruction` and bound relatively to the origin occurrence previously defined.

- **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.

4.1.7 Sample Rates

The `SampleRate` type serves as a specifier as part of a rule schedule. Sample rates determine how often the parameter set of a rule shall be sampled. We have implemented the following sample rates for the purposes of the MSRR prototype:

- **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.

4.1.8 Sample Points

The `SamplePoint` type serves as part of a rule schedule and are associated with a specific parameter of said rule schedule's expected behavior definition rule. A sample point specifies where in the location scope's range or ranges to take the samples. We have prototyped the following methods to serve our prototypes needs:

- **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.

4.1.9 Expressiveness Statement

The MSRR Measurement Policy language is prototyped to be a robust, general purpose solution for describing program behavior and specifying sample schedules. We have modeled the measurement system to be able to express as, simply as possible, a host of measurement policies that exist today in state of the art research. At the time of this writing, we believe that MSRR-PL is an effective tool to express the measurement needs and policies described in current remote attestation measurement works; this is explored further in chapter 6 and chapter 7. However, remote attestation and measurement is a rapidly growing area of research. With this in mind, we have designed the MSRR Policy Language to be easily extensible such that we can accommodate future measurement needs.

4.2 Implementation

We have implemented the MSRR Policy Language as a domain specific language in C++. This prototype policy language is a high-level format that has been implemented with a low-level backend that produces the MSRR Evidence Query Language commands for the MSRR Measurement System. This section describes in detail the features of MSRR-PL and the adaptation to the MSRR-EQL backend.

4.2.1 MSRR-PL to MSRR-EQL backend

The MSRR Policy Language provides a way of describing measurement policies at a high level. Ultimately, this high level form must translate to the low level invocations of a measurement system. For MSRR-PL, we have implemented a backend that produces MSRR-EQL queries to invoke MSRR Measurement System.

For a specific Sampling Schedule and a corresponding Expected Behavior Definition, MSRR-PL library can produce the EQL queries to register the periodic measurements that take the right samples and at the right frequencies. This is done by creating the a combination of EQL hook expressions, often nested, to instruct the measurer to sample according to the schedule.

Listing 4.11 and Listing 4.12 demonstrates a MSRR-PL policy and the corresponding MSRR-EQL queries, respectively. The example is a policy where a baseline measurement is made of some local variable x and subsequent measurements of x are expected to be greater than the baseline.

Listing 4.11 Baseline example: MSRR-PL policy definition

```
Policy policy;

// Build Behavior Definition

policy.behavior_definition
    .validation_functions["baseline_validation_function"] =
        new ValidationFunction(
            [](SampleSet samples) {
                int x_baseline = samples.getAsInt("x_baseline");
                int x_sample = samples.getAsInt("x_ongoing");
                return x_sample > x_baseline;
            }
        );

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

policy.behavior_definition.locations["x_location"] =
  new FileLineLocation("myfile.c", 7);

policy.behavior_definition.occurrences["baseline_occurrence"] =
    new FirstOccurrence("x_location");

policy.behavior_definition.occurrences["ongoing_occurrence"] =
    new FirstOccurrence("x_location");

policy.behavior_definition.parameters["x_baseline"] =
  new Parameter("x_feature", "baseline_occurrence");

policy.behavior_definition.parameters["x_ongoing"] =
  new Parameter("x_feature", "ongoing_occurrence");

policy.behavior_definition.rules["baseline_rule"] =
  new Rule(
    "baseline_validation_function",
    {"baseline_occurrence", "ongoing_occurrence"}
  );

// Build Sampling Schedule

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

policy.sample_schedules["default_schedule"]
  .rule_schedules["baseline_rule_schedule"] =
    new RuleSchedule(
      "baseline_rule", EveryIteration(),
      {
        FirstInstructionSamplePoint(),
        FirstInstructionSamplePoint()
      }
    );
```

Listing 4.12 Baseline example: MSRR-EQL expression

```
(seq
    (store "x_baseline" (measure (var "x")))
    (hook
        (reach (file_line_location "myfile.c" 7) true)
        (action (store "x_ongoing" (measure (var "x"))))))
```

4.2.2 Supporting Successor Measurements

Successor measurements are those that describe a measurement taken after the occurrence of a specific code region. Successor events can be useful for many different scenarios, as we have found this variations of this basic pattern reappear throughout our work. The following scenarios are both short examples of successor measurements:

- A rule that expects local variable X at location L1 to be greater than X at the next occurrence of location L1.

- A rule that expects some property to be true at the first occurrence of location L2, immediately after some occurrence of location L1.

Successor measurements are implemented in MSRR Policy Language using the NextOccurrence occurrence scope. The NextOccurrence is a OccurrenceScope that takes another OccurrenceScope as an input.

Listing 4.13 and Listing 4.14 demonstrates a MSRR-PL policy expressing this successor measurement relationship. In this example some local variable x and the immediate subsequent measurements of x are sampled and compared.

Listing 4.13 Successor example: MSRR-PL policy definition

```
Policy policy;

// Build Behavior Definition

policy.behavior_definition
    .validation_functions["successor_validation_function"] =
        new ValidationFunction(
            [](SampleSet samples) {
                int x_initial = samples.getAsInt("x_initial");
                int x_successor =
                    samples.getAsInt("x_successor");
                return x_initial > x_successor;
            }
        );

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

policy.behavior_definition.locations["x_location"] =
  new FileLineLocation("myfile.c", 7);policy

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

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

policy.behavior_definition.parameters["x_initial"] =
  new Parameter("x_feature", "initial_occurrence");

policy.behavior_definition.parameters["x_successor"] =
  new Parameter("x_feature", "successor_occurrence");

policy.behavior_definition.rules["successor_rule"] =
  new Rule(
    "successor_rule", {"x_initial", "x_successor"}
  );
```

```

// Build Sampling Schedule

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

policy.sample_schedules["default_schedule"]
  .rule_schedules["successor_rule_schedule"] =
    new RuleSchedule(
      "successor_rule", EveryIteration(),
      {
        FirstInstructionSamplePoint(),
        FirstInstructionSamplePoint()
      }
    );
```

Listing 4.14 Successor example: MSRR-EQL expression

```
(hook
 (reach (file_line_location "myfile.c" 7) true)
 (action (seq
    (store "x_initial" (measure (var "x")))
    (hook
        (reach (file_line_location "myfile.c" 7) false)
        (action (store "x_successor" (measure (var "x"))))))))
```

Chapter 5

Measurement Policy Generation

Utilizing the measurement system and measurement policy language discussed in the previous chapters, an expert tasked with developing application measurement solutions can save significant time and effort. The MSRR Measurement System provides the core measurement features which eliminate the need for the expert to build the measurement system from the ground up. Furthermore, the MSRR Policy Language gives the expert a high-level way of describing measurement policies so that they may better and more efficiently address those key parts of a given measurement system that are dependent on the target application.

These techniques provide a large improvement from the status quo, however, the task of writing measurement policies, even when armed with a good policy language, is difficult. The expert still must learn both the domain of each target application and the domain of measurement with regards to remote attestation. The extent, that such an individual is trained and skilled in these regards, is to the extent that the measurement systems they produce can adequately represent and evidence 'good state' and to do so in an efficient manner.

This section builds upon the tools provided in the previous chapters by prototyping a system to automate the generation of certain types measurement policies and, consequently, the process of producing tailored measurement solutions. Using state of the art static analysis techniques, we provide ways in which an expert can generate various types of rules and thus measurement policies. In the best case, the needs of a given attestation scenario will be satisfied by the automatically generated measurement system. In such cases, manual effort by an expert is eliminated. Even in cases where additional rule coverage is required, the automatic rules produced herein give the expert a blanket of initial coverage for 'free' thus minimizing their burden and allowing them to focus their efforts on the applications, and the features of said applications, that need the most trust-related attention.

This chapter describes the measurement policy generator that has been prototyped to utilize symbolic execution via SymInfer. First, we discuss the static analysis tools that we leverage. Then we describe how these tools have been incorporated by the MSRR suite. We demonstrate these techniques with an example of fully automated policy generation. And, finally, we discuss in more detail the practical impacts of these tools; their advantages and their limitations.

5.1 Symbolic Execution

The MSRR suite leverages the static analysis technique known as *symbolic execution* in order to discover expected states of the target application. These expected states enable us to form *program invariants* which can be scoped to form rules in the measurement policy.

Symbolic execution is a type of program execution where so-called *symbolic values* are used in the place of the concrete values that facilitate normal execution.

The symbolic values represent ranges of possible concrete values, come execution time [25].

Symbolic execution is non-deterministic. That is to say that, upon each program branch, symbolic execution explores both branches. Upon entering a branch, a *path condition* is applied which constrains the symbolic values to the new set of possible values.

Symbolic execution has been utilized to find state for bug finding and input generation, and to establish other properties of programs [5, 7, 17, 39, 43, 44, 51, 55].

Figure 5.1 demonstrates symbolic execution by displaying a trace, for each source-level instruction, the corresponding symbolic execution state information. The source level instructions are show in C on the right. The symbolic execution state is shown on the left. Each state has, starting from the left, the current path condition (PC) and then any new symbolic value assignments ('=') or conditional statement ('...?'). The symbolic values take the form i#. For example, variables *x*, *y*, and *z* start with unique symbolic values *i1*, *i2*, and *i3*, respectively. After the assignments to *z*, in either branch body, *z* begins to share the same symbolic value *i1* or *i2*, depending on which branch was taken.

The path condition starts out as *true*: in other words, 'no conditions'. After the branch of $x < y$, the path condition reflects the would-be concrete conditionals via the corresponding symbolic values $i1 < i2$ upon the true case. In the false case, the path condition becomes the complement: $i1 \geq i2$.

Symbolic execution has been leveraged to generate states for a target application [9], which can be leveraged to produce expected states and thus expected behavior. Symbolic execution allows for the production of program invariants [4].

Program invariants are properties that hold true for a specific range of a target application; these take the form of boolean propositions that are scoped to specific code instructions or instruction ranges.

Candidate program invariants can be derived via an analysis of the path conditions at critical points of the target application. The path conditions bound variables to possible ranges of target values, which can be used to form the performance envelope. The following section discusses how symbolic execution is used with other tools to produce a final set of invariants.

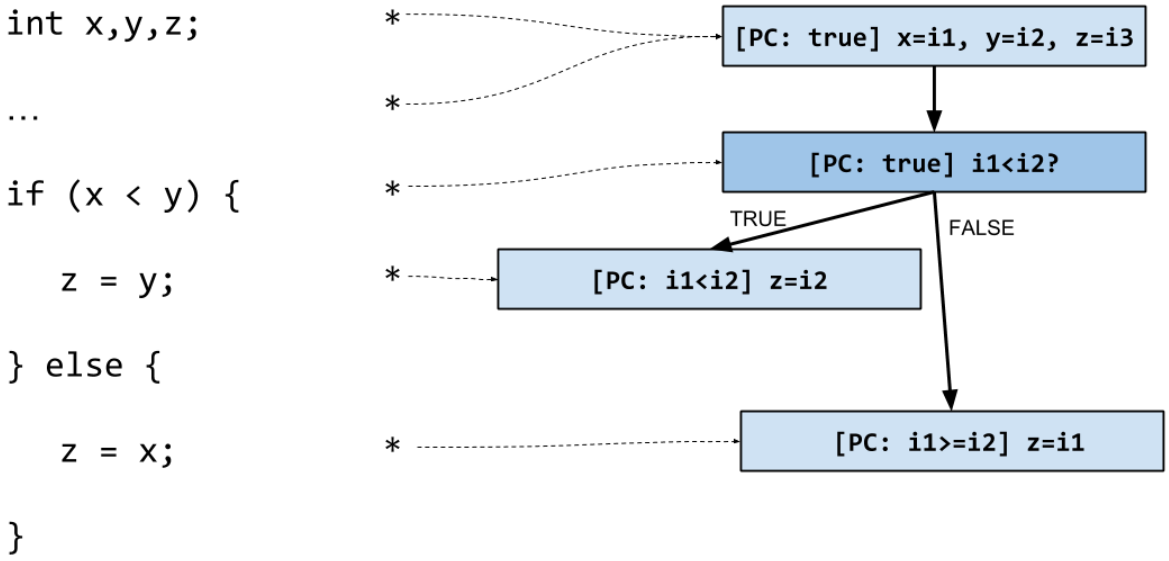
5.2 SymInfer, KLEE, & DIG

We prototype MSRR suite using the SymInfer tool which is designed to leverage symbolic execution to produce program invariants. SymInfer takes a program source as an input and will calculate program invariants at specific source-level instructions [36]. The results are boolean expressions which have been found to be true at the target code location for all possible inputs.

SymInfer uses an approach called *CounterExample Guided Invariant Detection* (CEGIR). CEGIR is a hybrid dynamic and static approach where candidate program invariants are inferred dynamically and then spurious candidates are refuted via static analysis. [35]

SymInfer implements CEGIR using symbolic execution via several tools. While the SymInfer was originally presented to work for Java programs using Symbolic PathFinder(SPF) [43], SymInfer also supports the symbolic execution tool KLEE [5] which we use in MSRR to produce symbolic states for C programs.

First, KLEE is used to produce a set of symbolic states. Then, concrete states are generated from the symbolic states using the Dynamic Invariant Generator



```
int x,y,z;

...

if (x < y) {

   z = y;

} else {

   z = x;

}
```

Figure 5.1. Symbolic state productions for a simple symbolic execution example

(DIG) [37]. From these concrete states, DIG infers expressive nonlinear candidate program invariants. Finally, SymInfer uses KLEE's symbolic states one more time to verify or refute the candidate invariants [36].

5.3 SymInfer to MSRR-PL Rule Translation

SymInfer leverages KLEE and DIG to produce program invariants for specified source level instructions. The conversion from these to MSRR-PL rules is a fairly straightforward process. The invariants take the form of a series of comparisons that can be conjoined to form the body of the MSRR-PL Validation Function. Then, said function's scoped parameters can be bound to the specific instruction, or instructions, upon which the SymInfer invariant was produced.

The remainder of this section will demonstrate MSRR's use of SymInfer with an example from the NLA micro-benchmark suite. The NLA benchmark suite is a collection of small math programs gathered by RodrÃguez-Carbonell and Kapur [36, 46, 47, 48].

Source Program We will use ‘cohendiv.c’, an application which computes integer division. Cohendiv is part of the NLA micro-benchmark suite.

Listing 5.1 Rule translation example: cohendiv.c target application

```
int mainQ(int x, int y){
    //Cohen's integer division
    //returns x % y

    assert(x>0 && y>0);

    int q=0;
    int r=x;
    int a=0;
    int b=0;
    while(1) {
        ////%%%traces: int x, int y, int q, int r
        if(!(r>=y)) break;
        a=1;
        b=y;

        while (1){
            //assert(r>=2*y*a && b==y*a && x==q*y+r && r>=0);
            //%%%traces: int x, int y, int q, int a, int b, int r
            if(!(r >= 2*b)) break;

            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    //assert(r == x % y);
    //assert(q == x / y);
    //assert(x == q*y+r);
    //%%%traces: int x, int y, int r, int q, int a, int b
    return q;
}
```

```

int main(int argc, char **argv){
     mainQ(atoi(argv[1]), atoi(argv[2]));
     return 0;
}
```

The source of cohendiv is shown above. Special comments have been inserted as annotations for SymInfer. The comments which start '%%%traces' indicate to SymInfer to calculate a program invariant at said location. Additionally, each such comments includes the local variables via C type and identifier, to be considered in the formation of the program invariant.

SymInfer Output For cohendiv.c, we can run SymInfer to produce the following output file:

Listing 5.2 Rule translation example: SymInfer output

```
*** 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
```

Policy Generation Output From the SymInfer output, we generate a rule for each invariant listed.

As discussed in chapter 4, a rule in MSRR-PL is essentially a validation function associated with scoped parameters as inputs. The scoped parameters are all target features which are scoped to a specific location and relative occurrence.

As such, the validation function becomes the conjunction of the set of numerical comparisons that comprises the invariants for the target line. Moreover, the parameters for such a validation function are simply the local variables using the same source-level identifier that are shown in the SymInfer output. The parameters are scoped to the specific file and line in said output.

The validation function will take the following form for the first invariant. First invariant:

Listing 5.3 Rule translation example: First invariant

```
25: a*y - b == 0, q*y + r - x == 0, -b <= -1, b - r <= 0,
    r - x <= 0, -y <= -1
```

MSRR-PL Validation function:

Listing 5.4 Rule translation example: MSRR-PL 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;
            }
        );
```

The scoped parameters will take the following form:

Listing 5.5 Rule translation example: features, scopes, parameters

```
policy.behavior_definition.features["a_feature"] =
    new VariableFeature("a");

policy.behavior_definition.locations["location_1"] =
    new FileLineLocation("cohendiv.c", 25);

policy.behavior_definition.occurrences["occurrence_1"] =
    new OriginOccurrence("location_1");

policy.behavior_definition.parameters["a_parameter"] =
    new Parameter("a_feature", "occurrence_1");
```

The full rule for the first invariant is below:

Listing 5.6 Rule translation example: full MSRR-PL policy

```
Policy policy;

// Build Behavior Definition

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;
            }
        );

policy.behavior_definition.features["a_feature"] =
    new VariableFeature("a");
policy.behavior_definition.features["b_feature"] =
    new VariableFeature("b");
policy.behavior_definition.features["q_feature"] =
    new VariableFeature("q");
policy.behavior_definition.features["r_feature"] =
    new VariableFeature("r");
policy.behavior_definition.features["x_feature"] =
    new VariableFeature("x");
policy.behavior_definition.features["y_feature"] =
    new VariableFeature("y");

```

```
policy.behavior_definition.locations["location_1"] =
    new FileLineLocation("cohendiv.c", 25);


policy.behavior_definition.occurrences["occurrence_1"] =
    new OriginOccurrence("location_1");

policy.behavior_definition.parameters["a_parameter"] =
    new Parameter("a_feature", "occurrence_1");
policy.behavior_definition.parameters["b_parameter"] =
    new Parameter("b_feature", "occurrence_1");
policy.behavior_definition.parameters["q_parameter"] =
    new Parameter("q_feature", "occurrence_1");
policy.behavior_definition.parameters["r_parameter"] =
    new Parameter("r_feature", "occurrence_1");
policy.behavior_definition.parameters["x_parameter"] =
    new Parameter("x_feature", "occurrence_1");
policy.behavior_definition.parameters["y_parameter"] =
    new Parameter("y_feature", "occurrence_1");

policy.behavior_definition.rules["rule_1"] =
    new Rule(
        "validation_function_1",
        {
            "a_parameter", "b_parameter", "q_parameter",
            "r_parameter", "x_parameter", "y_parameter"
        }
    );

// Build Sampling Schedule

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

policy.sample_schedules["default_schedule"]
    rule_schedules["rule_1_schedule"] =
    new RuleSchedule(
        "rule_1", EveryIteration(),
        {
            FirstSamplePoint(), FirstSamplePoint(),
            FirstSamplePoint(), FirstSamplePoint(),
            FirstSamplePoint(), FirstSamplePoint()
```

```
          }
     );
```

5.4 Discussion & Limitations

In the best case, the requirements for trust are such that this technique is sufficient to automatically generate measurement policies that can be used to eliminate the expert and their manual effort from the process of writing application specific measurers entirely. In such cases, the entire process of generating an application specific measurement system is automatic.

However, in some cases, the types of measurement policies that the automatic generation system produces, and the *rule coverage* of such, may not be sufficient for the specific trust requirements of a given remote attestation system. In such, cases the MSRR policy generation tool can be used to get initial rule coverage for free and allow the expert to focus on the other areas of the policy.

This may be the case for certain types of policies because of the nature of human written policies contrasted with those automatically discovered through invariant inference. The human-written policies tend to be conceptually driven in nature. These will take forms similar to the likes of human-written unit tests which attempt to encapsulate the intentions or purpose of a specific component of the software. These such policies might describe high-level notions such as, in a game of Chess, what type of move is considered legal, in accordance with the rules of a standard game of chess. This example and others are discussed in chapter 7.

In contrast, the automatically generated policies will take non-conceptual forms. This process will be better at finding low-level relationships between target features. These sorts of policies are better at getting lots of ground coverage that are agnostic of high-level conceptual and program-purpose related features.

All things considered, we believe that automatic techniques such as the one we employed in this chapter are be best suited for garnering high degrees of rule coverage at a very low cost. Most systems have a host of heterogeneous applications that would otherwise be prohibitively expensive, out of the sheer enormity of scope, to get meaningful rule coverage from the manual effort of experts. This automatic technique is excellent in such cases to provide high degrees of coverage and to allow experts to go either completely uninvoked or to focus their efforts on the most trust critical subsets of the target entity.

Chapter 6

Suite Fitness & Performance Benchmarking

The MSRR measurement suite is a set of tools designed to work in unison to dramatically reduce the time and effort spent generating tailored measurement solutions for arbitrary target applications. To this end, we believe we have been successful in bringing together state of the art techniques along with new strategies to facilitate such improvements. Furthermore, we believe that the tools and techniques we provide are reasonably applicable, expressive, and efficient.

This chapter will discuss the fitness of the MSRR tools, both in regards to their ability to eliminate manual work and, where appropriate, the extent to which the tools do so efficiently, both in their own performance and the degree to which they minimize impact on the performance of the target application. To this end, we offer several experiments which comment upon the fitness of the various components of the MSRR approach.

This chapter will be organized as follows. First, we will discuss the goals and metrics for which such a system should be judged. The following sections will each describe an experiment targeting the aforementioned goals, the results, and any related discussion and analysis. The final section will summarize the findings, both the advantages and the limitations for the MSRR suite.

6.1 Goals & Metrics

The MSRR Measurement Suite is designed to lessen the manual effort required by would-be experts in writing measurement systems. To this end, the software components must be able to perform the various tasks pursuant to the goals, and the language components must be able to express the desired range of functions and features pursuant to the same. Moreover, for the software components, there is also the question is how efficiently do they perform their tasks or express their content. Likewise, the language components are better to the extent the are efficient in usability, clarity, and conciseness.

In this section, we will address both the measurement system and the measurement policy language discussed in this work in regards to their goals for fitness.

General Purpose Measurement System The MSRR Measurement System is the component that provides the core measurement features. It is the software piece that is directed by the attestation system in order to control and record execution state of the target application.

Such a dynamic measurement system should actively maintain trust throughout the life of a target application. Such a measurer must be lightweight and as unintrusive to the target applications as possible.

The bottom line for the MSRR Measurement System is to ensure that it can capture state, and do so with a minimum or at least reasonable overhead to the target application. In the following sections, we present experiments which target specifically the overhead of the measurement system in various contexts. Moreover, we record performance data of the measurement system under different loads, to express the performance impact generally. We also take detailed data to explore the specific performance impacts of some of the key measurement types, including the snapshot measurement.

Measurement Policy Language The MSRR Policy Language is used to write measurement specific policies so that a measurement system may be driven accordingly. Such a policy language has two main roles, as discussed in detail in chapter 4: (A) it must encapsulate the expected behavior of the target application and (B) it must describe a schedule for the runtime sampling of the target application, in order to evidence said expected behaviors.

The question of 'what constitutes a good or fit policy language' is an open research question, especially at anything more than a very high level. This is primarily due to the relative infancy of the literature with regards to dynamic measurement for arbitrary applications. As such, the kinds of policies themselves that will prove most useful in the near future and beyond is only starting to come into view. Furthermore, even with a good understanding of what a quality policy language should look like, the task of evaluating said quality is daunting given that there are few policy languages to compare to, and none that suite the general case, as does MSRR-PL.

That said, like with any specification language beyond the realm of measurement, the following attributes are universally desirable: A more expressive language is better; that is, we want to be able to, ideally, express all possible good measurement policies. Furthermore, we want to express such while also being both concise and easy to understand.

In an attempt to best demonstrate MSRR-PL's success and failings in regards to these universal language values, we have selected several example policies. Using these examples as language benchmarks, we analyze possible MSRR representations of them and discuss the efficiency and ease to which they can be written. Moreover, we provide quantitative metrics where possible and analyze qualitatively in either case.

6.2 Experimental Methodologies

We conduct our experiments on a system running 64-bit Fedora 24 with 32 GB of memory and quad-core Intel Xeon 1.8 Ghz processor. We employ custom micro-benchmarks, the Non-Linear Arithmetic (NLA) micro-benchmark suite [36, 46, 47, 48], and programs from the SPEC CPU 2006 benchmark suite with the reference data sets [21]. We compiled the benchmarks with the -g option to produce the DWARF symbols for use by our measurement system. We have used default settings for all other options. Unless otherwise stated, each benchmark-configuration is executed 10 times, and the average program runtime is used.

6.3 Experiment 1

Our first experiment is designed to evaluate MSRR Measurement System's overhead when it is *attached* to the target application and is ready to collect measurements, but receives no requests from the attestation system during the process life-time. In this scenario, we found that MSRR does not impose any discernible overhead that is within the margin of error for any of our benchmarks.

6.4 Experiment 2

The second experiment uses a simple micro-benchmark (computing the Fibonacci sequence) to measure the cost of individual measurement events. We create a timed monitoring measurement (using `delay`) to sample the program call stack (`call_stack`), a specific machine register (`reg`), and a stack memory variable (`mem`) every 10 msec. We create another event-based continuous measurement to measure the cost of the `hook` mechanism. The `hook` stops the program at a specified program location and immediately returns without collecting any measurement.

The experimental setup is designed to collect approximately 22,000 samples of any one measurement type during a single program run. We also create a timed event to measure the overhead of the snapshot utility that calls `snap` every 10,000 msec.

These measurement functions are described in detail in subsection 3.1.5. The snapshot measurement is used for the optimization which is discussed in subsection 3.1.8.

The *baseline* executes the micro-benchmark without any measurement. Each *active* run activates a single measurement type during program execution. The time difference between the *active* and *baseline* program runs, divided by the number of events invoked gives us the estimate of the cost of each event. We find that the `call_stack`, `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, respectively, on our system.

The cost of some events, especially `call_stack` and `snap`, may vary depending on the client program's call stack depth and memory usage.

6.5 Experiment 3

The third experiment evaluates the overhead imposed by the MSRR framework when sampling the entire call stack of a user-application at different measurement frequencies.

The experiment performs the following sequence of steps.

1. Start the target process with LD_PRELOAD of a custom library which halts the program before entry into the target program's `main` function.

2. The measurer daemon is launched.

3. The attester program is launched.

4. The attester initiates communication with the measurer daemon.

5. The attester directs the measurer to begin monitoring the call stack at the configured frequency.

6. The attester directs measurer to modify the hold flag to allow the program to enter main and commence normal execution.

7. The measurer takes periodic measurements of the call stack until the program terminates. Measurements are recorded into the measurement store.

*Note the measurer's ability to modify the data of the target is reserved to evaluation scenarios like these. In a classical remote attestation scenario, the application can only be temporarily halted and sampled by measurer, as described in subsection 3.1.1.

Measurements are collected at periodic intervals of 100 msec, 1,000 msec, 10,000 msec, and at every system call.

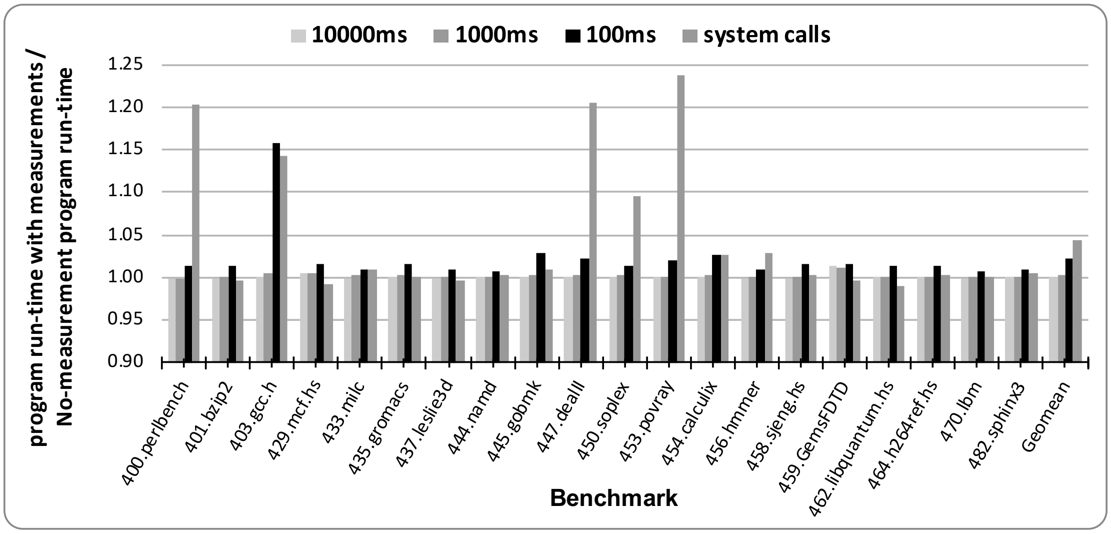


Figure 6.1. MSRR overhead when invoked to periodically sample the application call stack

Figure 6.1 shows the ratio of the program runtime when measurements are taken for various configurations to the program runtime with no measurer attached. We found that the measurer imposes an overhead of 0.08%, 0.25%, 2.14%, and 7.95% for call stack measurements taken every 10,000 msec, 1,000 msec, 100 msec, and at all system calls, respectively and on average (geometric mean) over all benchmark programs. The standard deviations were small relative to their means. The average standard deviation was 0.38% and all standard deviations fell in the range of 0.01% and 3.65%.

Distinct benchmarks have both a different system call invocation rate and different average call stack depths. These differences cause the large variation in the overhead imposed by MSRR for call stack samples at system call sites for different programs. The timer-based events trigger the measurements at a uniform rate (100 msec, 1,000 msec, or 10,000 msec) for all benchmarks. For these timer-based experiments, the largest overhead was on benchmark 403.gcc at a measurement period of 100 msec. The average execution time for 403.gcc was 115.7% of the execution time without any measurement. We found that the higher overhead is mainly because 403.gcc routinely has higher call stack depths than most other benchmarks.

The delay duration represents a tradeoff between performance and accuracy of trust based inferences. Id est, more frequent measurements decreases the likelihood that transient corruptions, as may be the result of an attack, are not represented by the evidence. Even with very high frequency measurements, our measurer performs reasonably with small overhead.

6.6 Experiment 4

This section exercises the policy language and policy generation components using the Non-Linear Arithmetic (NLA) micro-benchmark suite. The NLA benchmark suite was originally used by SymInfer to demonstrate its own capabilities to infer program invariants from a host of programs.

For each program in the NLA suite, SymInfer infer can generate program invariants. And, we have in turn used MSRR to generate MSRR-PL policies.

As such, we can reasonably infer that MSRR can generate policies from all SymInfer outputs, to the extent that SymInfer's own selection of the NLA benchmark suite as a representative set is accurate.

Using the policies generated from the NLA micro-benchmark suite, we performed a live experiment with the MSRR measurement system in order to demonstrate the overhead of the autogenerated policies. Specifically, the fourth experiment evaluates the overhead imposed by the MSRR framework when sampling as directed by the autogenerated policies for each NLA benchmark, and at different measurement frequencies which have been encoded via different sampling

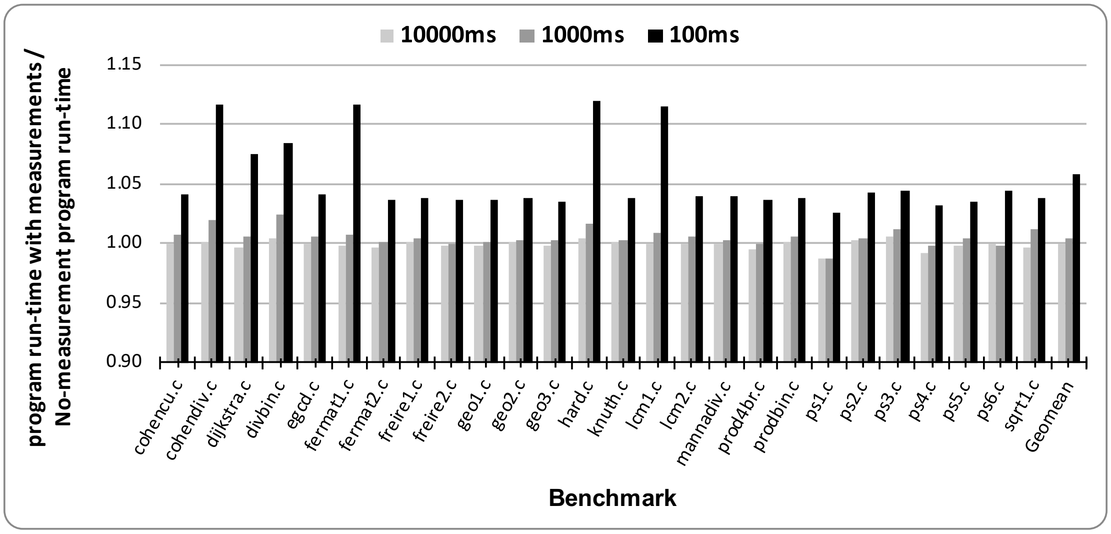


Figure 6.2. MSRR overhead when invoked periodically to sample call stacks for automatically generated policies

schedules.

The experiment performs the same sequence of steps as Experiment 3, in section 6.5. Measurements schedules are designed to collect samples at periodic intervals of 100 msec, 1,000 msec, and 10,000 msec.

Figure 6.2 shows the ratio of the program runtime when measurements are taken for various configurations to the program runtime with no measurer attached. We found that the measurer imposes an overhead of 0.53% and 5.29% for call stack measurements taken every 1,000 msec and 100 msec, respectively and on average (geometric mean) over all benchmark programs. For a period of 10,000 msec, the overhead was statistically insignificant. The standard deviations were small relative to their means. The average standard deviation was 0.67% and all standard deviations fell in the range of 0.13% and 3.51%.

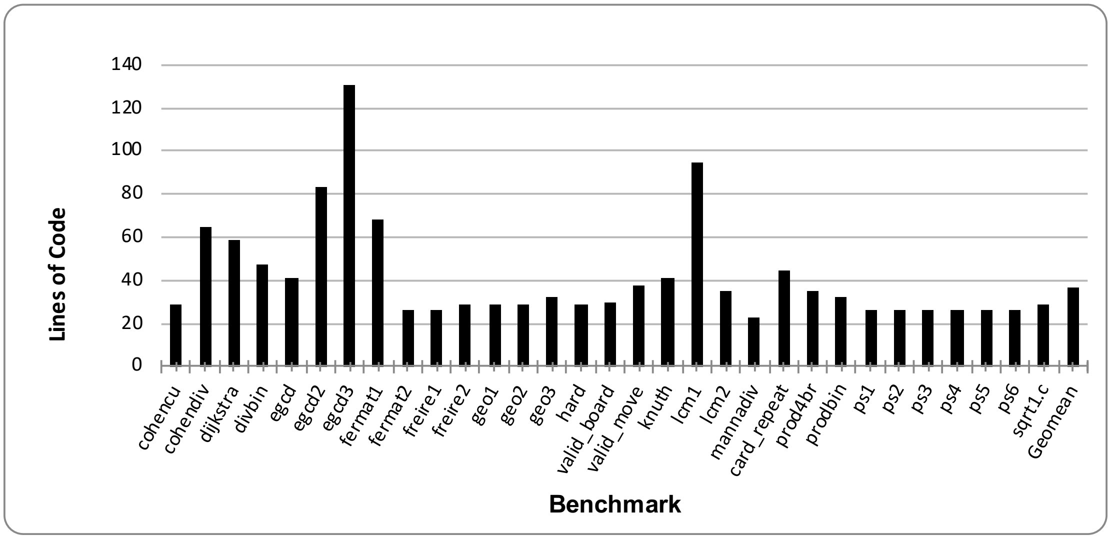


Figure 6.3. Number of lines of code for the benchmark measurement policies written in the MSRR Policy Language

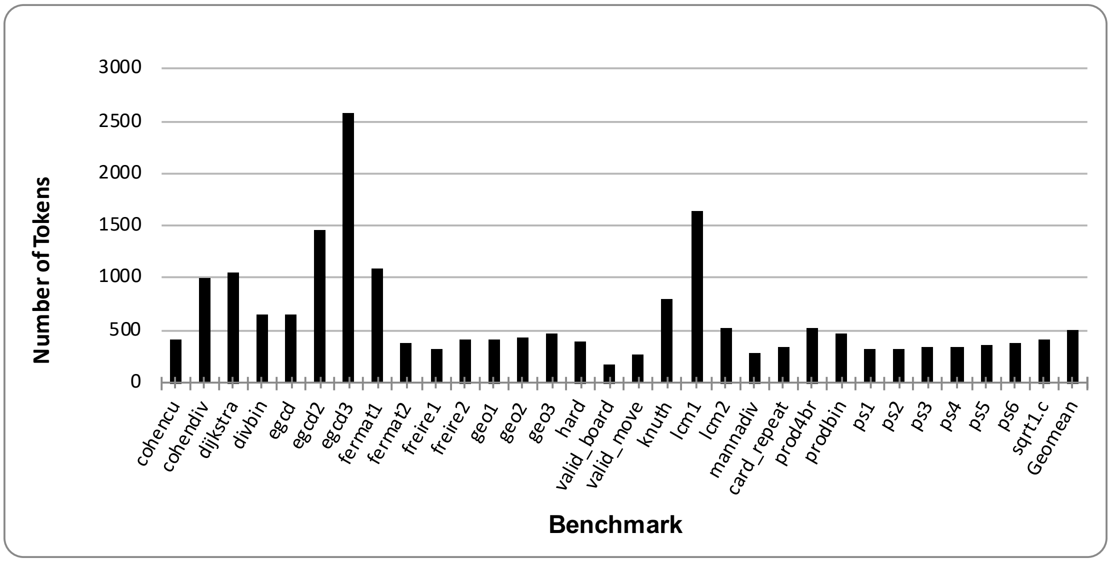


Figure 6.4. Number of tokens for the benchmark measurement policies written in the MSRR Policy Language

6.7 Experiment 5

In this section, we have produced several measurement policies in the MSRR Measurement Policy Language. Using these examples, we perform industry standard code metrics to find insight into the qualities of the MSRR-PL policies.

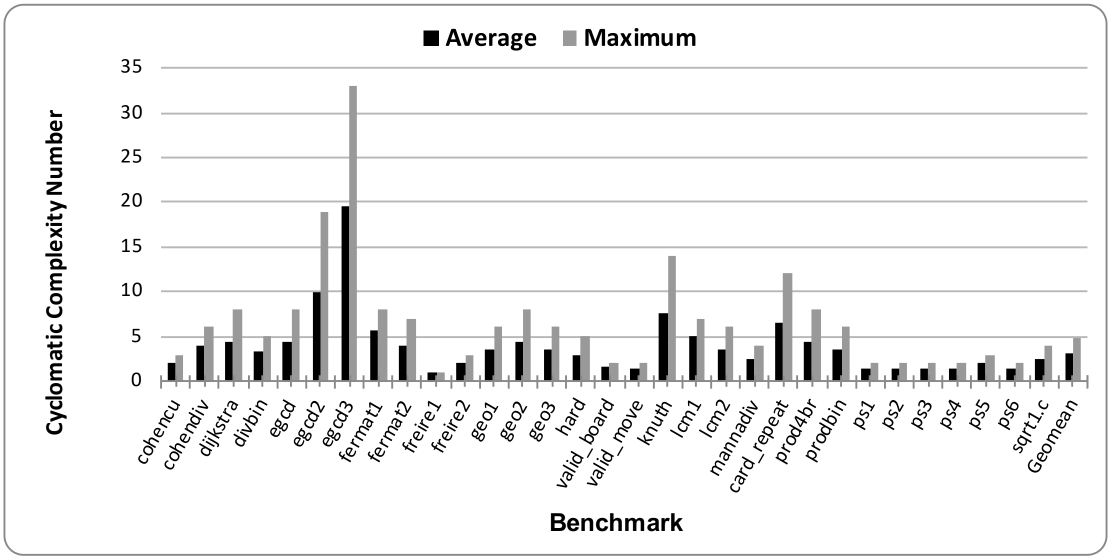


Figure 6.5. Average and maximum cyclomatic complexity, by method, for the benchmark measurement policies written in the MSRR Policy Language

We have curated a varied selection of measurement scenarios, and ultimately policies. We borrow a policy written for the Java Measurement Framework (JMF) for the bluffin-muffin Texas Hold'em card game simulator [56, 59]. We utilize the policies generated by the MSRR policy generation tool, for each of the 27 benchmarks in the NLA micro-benchmark suite. We employ two custom examples for the DreamChess program [57], which are described in detail as case studies in chapter 7.

For each benchmark policy, we write or rewrite the policy in MSRR-PL using industry standard best practices. Then, we report following industry standard metrics for measuring code complexity:

- Lines of Code - The number of lines of code in the policy language definition.
- Token Count - The number of C++ tokens used in the policy language definition.

- Cyclomatic Complexity Number (CCN) - A control flow graph derived complexity estimation metric [31].

Cyclomatic complexity is code metric used in the industry to increase testability and maintainability of software code. Method complexity measures such as cyclomatic complexity are highly related to the likelihood of error presence in a software [50].

A limit of a cyclomatic complexity number (CCN) 10 is suggested for software development compliance. McCabe suggests that code which exceeds the limit should either be refactored or a rationale should be provided as to why the high complexity is appropriate [60].

The cyclomatic complexity is calculated analyzing the source code's control flow graph and by counting the number of edges, nodes, and exit points therein. Cyclomatic complexity is calculated with the following formula [27, 31]:

$$M = E - N + 2P$$

M is the complexity number. E is the number of edges in the control flow graph. N is the number of nodes. P is the number of exit points.

In our experiment, cyclomatic complexity was calculated at a per-method level. Figure 6.5 shows the average and maximum per-method cyclomatic complexity number for each measurement policy.

We found that the MSRR-PL policies had an average (geometric mean) of 36.9 lines of code for all benchmark policies and 505.8 tokens for all benchmark policies. The lines of code and number of tokens scale linearly with respect to the number of parameters of the validation function. We found that the MSRR-PL definitions had an average (geometric mean) of 3.14 CCN per-method averages and 4.85 CCN per-method maximums.

6.8 Findings Summary

The MSRR Measurement Suite is designed with the express purpose of reducing and eliminating, where possible, the significant manual effort required to develop tailored application-specific measurement solutions. To this end, we believe we have succeeded with our prototypes. Not only does our system lessen manual effort, for the various tasks on the application measurer development pipeline, the evidence suggests that we do so efficiently in regards to the simplicity of our solutions and low impact of the tools.

In this section, we will now restate the goals and summarize the findings.

General Purpose Measurement System A measurement system should be lightweight and as unintrusive to the target applications as possible.

Our findings show that the MSRR Measurement System can capture state with a reasonable overhead to the target application for a variety of workloads, both in policy type and sample rates. Each of the non-snap EQL features imposed a very low overhead when measured individually. And, for even higher frequencies of measurement, the slowdown was reasonable.

The snapshot measurement, which is proposed for the optimization strategy discussed in subsection 3.1.10, will likely be most useful for data intensive applications, such as those that interface with relational databases where massive batch measurements may be required to verify rules involving mass data relationships. In such cases, we anticipate that the cost of direct measurements will easily exceed the threshold to make the snapshot optimization the preferred approach.

Measurement Policy Language A measurement policy should express as many types of measurement policies as possible, while also balancing with it conciseness and maintainability.

Based on the representative subset of benchmark policies that we have curated, the measurement policy language prototype is able to express polices of the nature of those that have surfaced as of yet in the remote attestation community. Furthermore, these policies have been demonstrated to have low complexity on each of the common industry code metrics that we measured, which correlates to higher maintainability and lower likelihood of error introduction upon refactor.

The average cyclomatic complexity fell are below the maintainability limit of 10 originally proposed by McCabe [60]. The few examples that exceeded the threshold the were automatically generated via the measurement policy generator. Those examples had validation functions with many numerical comparisons coming from the SymInfer produced program invariants. The higher complexity readings in this case introduces no concerns for maintainability because of the fact that these validation functions were produced automatically and can easily be recomputed automatically upon target application refactor.

Our findings also indicate that much of the code across all of our policies follow simple patterns. This is promising in that manual policy writing can be aided further via policy template generation tools which produce much of the repetitive code structures for the developer. Furthermore, advanced language features such as forms of ‘syntactic sugar’ that can decrease the verbosity and repetitiveness of the policy declarations would decrease the source size of the policies and increase development velocity.

Chapter 7

Examples and Case Study

In this chapter we present a few examples to illustrate the full usage of the MSRR suite, in the context of remote attestation and using real world applications.

7.1 Simple Attestation System

This section describes a very simple attester/appraiser software that can initialize an MSRR-PL policy and then utilize it by periodically retrieving measurements and verifying the rules.

In a real-world remote attestation scenario, the appraiser would be separated from the attestation system and would reside on a remote machine. As such, instead of making a decision on the attester's own, the attester would be sending measurements as evidence back to the appraiser for the trust-based decision making.

Listing 7.1 Example of a simple attester utilizing a MSRR-PL policy

```
// build a policy like one of the earlier examples
Policy policy = build_policy();
```

```

// connect to the measurer
Measurer measurer = new Measurer( ip_address , port );

// apply the sampling schedule to the measurer
measurer.apply_schedule(policy.schedule);

// begin main appraisal loop
while(true) {

    // wait before retrieving samples
    sleep(1000);

    // send a retrieve samples so far
    SampleSet samples = measurer.retreive_samples();

    // apply rules to the samples
    vector<ApplicationResult> ars =
        policy.apply_rules(samples);

    // handle the results accordingly
    cout << "Rule Applications: ";
    for (ApplicationResult ar : ars) {
        cout << (ar.pass ? "PASS" : "FAIL") << ",";
    }
    cout << "\n";
}
```

In Listing 7.1, the build_policy function can be replaced with any policy, such as those seen in previous chapters or in the following case study.

7.2 DreamChess Case Study

In this section, we present a detailed use case of the MSRR suite for a real world scenario. To this end, we have selected an application that is illustrative in nature and easy understand, both for the purposes of understanding its design concerns and how such concerns relate to the questions of expected behavior and establishing trust in the context of remote attestation. This application is DreamChess, a software designed to simulate chess games [57].

For this application, we discuss some key properties of the application that we wish to verify against potential malicious modification and attack. For each property, we demonstrate how our system can be used to describe the properties in measurement policies. And, for each policy, we show how the policy is used at runtime to invoke the measurement system to detect any aberrations from the expected values and relationships.

7.2.1 DreamChess Scenario

We employ *DreamChess*, which is an open source chess game for Windows, Mac, and Linux. DreamChess ships with the chess engine Dreamer [57]. The trust framework is tasked with ensuring that the game of chess is being played correctly and fairly. This is an example of an application where an attestation protocol would be designed and measurement system employed to evidence said correctness of play.

For context, you may think of the appraiser in this case as an agent or component of some "gaming authority" which acts as a referee of online chess games. The goal for this "referee" is to establish that games are played fairly. To make the scenario interesting, money, prestigious chess titles, and even the gaming service provider's reputation could all be at stake.

DreamChess describes the game board (see Figure 7.1), in a structure called *board*. The structure contains an integer array *square* of size 64, to represent the spaces of the board. The first eight elements of the array correspond to the first row of spaces on the board, which are {a1, b1, c1, d1, e1, f1, g1, h1}. The

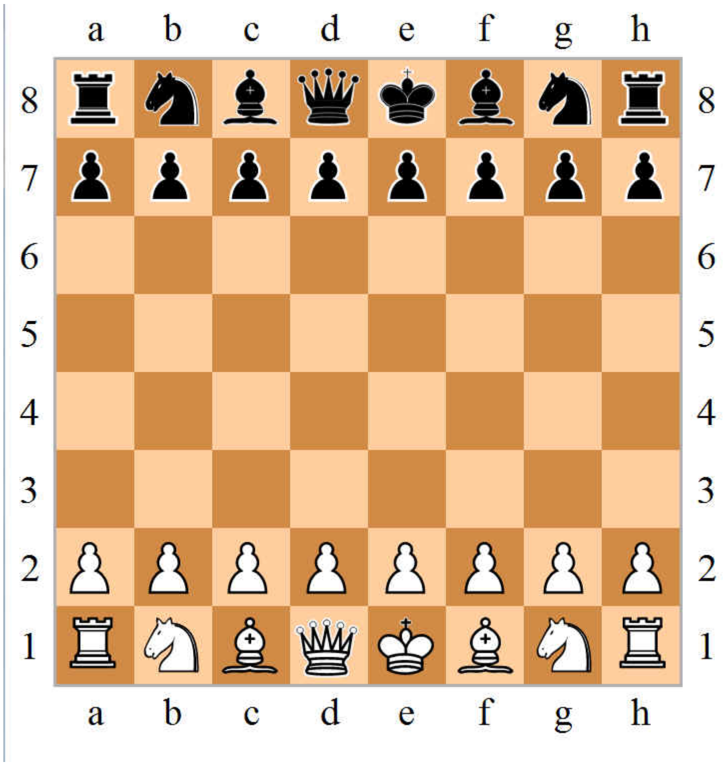


Figure 7.1. Starting configuration of a standard chess board with row and column labels

next 8 elements correspond to the next row, {a2, b2, c2, d2, e2, f2, g2, h2}, and so on. There is a unique integer value reserved for each game piece and color combination, and an additional value reserved for a blank space. See Listing 7.2 for the flags and the C definition of *board*.

Listing 7.2 DreamChess examples: relevant code regions [57]

```
#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;
```

7.2.2 MSRR-PL Policies for DreamChess

For DreamChess, we would like to describe some performance envelope and encapsulate that into an MSRR-PL policy. To do this, we can approach it in a similar manner as does a unit tester does when they have specific units of a piece of software upon which they would like to establish rules.

At the center of the operation of the chess simulation is the method where the player's "moves" are applied. In this method, the board changes state from one play state to another which makes this method a reasonable place to begin describing MSRR-PL rules.

The remainder of this section will identify properties for this critical region of DreamChess that we would like to encapsulate in MSRR-PL rules. These properties will be considered our 'goals'. For the goals, we will define a rule in MSRR-PL. And at the end of the section, a complete policy will be shown.

7.2.3 Rule 1 - Is the Chess Board in a Valid State?

The referee/appraiser may wish to determine that the board itself is a valid chess board, at any given time. For this section, we will make encapsulating this property in an MSRR-PL rule our goal.

To achieve this, we must answer the question of ‘What is a valid chess board?’ We must then answer ‘How are such boards represented in the DreamChess source?’ And finally, we must establish a rule that expresses the relationships of DreamChess’s source features to encapsulate this valid-boardness property.

So, what is a valid chess board, generally speaking? A valid chess board abides by the rules of a standard chess game and in truth is rather complicated. In a game of chess, pieces move around on the board. Some may eventually be removed from play. The Kings must be in play and not in ‘check’ so long as the game persists. Some pieces can never reach certain squares. Some pieces can transform into others.

The bottom line is that the notion of a valid chess board, to a high degree of accuracy, is counterproductively complex for the purposes of this example. However, a simplified definition of valid suffices to get the benefits of the MSRR-PL demonstration. That said, for this MSRR-PL rule, we will simplify the reality and thus the validation function logic to say the following: *A chess board shall be considered ‘valid’ if there are no more than 8 pawns of black and no more than 8 pawns of white.*

Listing 7.3 Valid board example: MSRR-PL policy definition

```
Policy policy;

// Build Behavior Definition

policy.behavior_definition
```

```
    .validation_functions["board_validation_function"] =
        new ValidationFunction(
            [](SampleSet samples) {
                vector<int> squares =
                    samples.getAsVector<int>("squares_parameter");

                return !(
                    count_if(squares, BLACK_PAWN) > 8 &&
                    count_if(squares, WHITE_PAWN) > 8
                );
            }
        );

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["make_move_occurrence"] =
    new OriginOccurrence("make_move_location");

policy.behavior_definition.parameters["squares_parameter"] =
  new Parameter("squares_feature", "make_move_occurrence");

policy.behavior_definition.rules["valid_board_rule"] =
  new Rule("board_validation_function", {"squares_parameter"});


// Build Sampling Schedule

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

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

Listing 7.3 demonstrates this rule. The validation function has the logic which encapsulates the expected states of a valid board. The remaining code, as with previous examples, is in charge of setting up the MSRR-PL Scoped Parameter of the board (technically the ‘square’ int array within struct board), associating it with the validation function, and finally establishing the sampling schedule.

A measurement request and response for this rule would look as follows.

Query	(retreive)

JSON-RPC	`{`
	`    "jsonrpc" : "2.0",`
	`    "params" :`
	`    {`
	`        "type" : "retreive_expr"`
	`    },`
	`    "method" : "eval",`
	`    "id" : 4`
	`}`

Result	(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
	)))

JSON-RPC	{
	    "jsonrpc": "2.0",
	    "result":
	    {
	        "samples" : [
	            {
	                "data" : {
	                    "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
	                    ],
	                    "type" : "int_value"
	                    "array" : **true**,
	                },
	                "label" : "board",
	                "occurrence" : -1,
	                "type" : "sample_result"
	            }
	        ],
	        "type" : "sample_set_result"
	    },
	    "id": 4
	}

7.2.4 Rule 2 - Is the Chess Move Valid?

A referee/appraiser might also be interested in ensuring that each and every move made during play is valid. For this section, we will make encapsulating the behavior of a valid move in an MSRR-PL rule our goal.

To this end, a policy can be constructed to take a baseline board measurement at the start of play and then a new measurement of the board upon the conclusion of each *move* operation. The difference between each successive board state will determine whether or not the move was valid.

Like before, to write a rule we must answer a few questions. In this case: What is a valid move in the game of chess, generally? How is this represented in the DreamChess source? How can we write a rule that relates the source features of DreamChess to distinguish valid moves from invalid moves?

As with the valid board concept, the concept of a valid move is more complex than is worth representing with complete accuracy in this demonstrative exercise. Chess describes an initial set of moves that can be made for each piece type, which are varied. Moreover, there are special exceptions such as castling and piece promotion. Consequently, as with the previous example, we will use a simplified reality to get the maximum benefit of the demonstration without needing to dive deep in the idiosyncrasies of chess.

For the purposes of this rule, we shall define a valid move as follows: *A valid move shall be defined as one that leads to a board difference in exactly two spaces.* The intuition here is that, in basic move scenarios, a piece is removed from its starting space and then replaces the contents of another space.

As such, the rule can be described as follows:

Listing 7.4 Valid move example: MSRR-PL policy definition

```
Polipolicycy policy;

// Build Behavior Definition

policy.behavior_definition
    .validation_functions["move_validation_function"] =
        new ValidationFunction(
            [](SampleSet samples) {
                int[] squares_initial =
                    samples.getAsVector<int>("initial_parameter");
                int[] squares_final =
                    samples.getAsVector<int>("successor_parameter");

                int[] squares_difference = subtract_array(
                    squares_initial, squares_final
                );

                return count_nonzero(squares_difference)==2;
                }
        );

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");

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

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

policy.behavior_definition.rules["valid_move_rule"] =
  new Rule(
    "move_validation_function",
    {"initial_parameter", "successor_parameter"}
  );

// Build 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()}
    );
```

The command to register the monitoring measurement is below.

<table>
<tr><td>Query</td><td>(retreive)</td></tr>
<tr><td>JSON-RPC</td><td>

```
{
    "jsonrpc" : "2.0",
    "params" :
    {
        "type" : "retreive_expr"
    },
    "method" : "eval",
    "id" : 4
}
```

</td></tr>
</table>

Result	(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 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)))

JSON-RPC

```
{
    "jsonrpc": "2.0",
    "result":
    {
        "samples" : [
            {
                "data" : {
                    "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
                    ],
                    "type" : "int_value"
                    "array" : true,
                },
                "label" : "board_initial",
                "occurrence" : -1,
                "type" : "sample_result"
            },
            {
                "data" : {
                    "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
                    ],
                    "type" : "int_value"
                    "array" : true,
                },
                "label" : "board_final",
                "occurrence" : -1,
                "type" : "sample_result"
            }
        ],
        "type" : "sample_set_result"
    },
```

Chapter 8

Future Work

The MSRR measurement suite described in this paper is an effort to reduce the costly manual effort required of experts which, by and large, prohibit the widespread adoption of application-level measurement system in trusted computing. As we have demonstrated, progress has been made on a several fronts herein; however, this is area of study is in its infancy and there is much room to improve.

Here are a few avenues for future work that we feel are most imminent and compelling.

Improvements to the Policy Language We envision a more ideal policy language implemented in a standalone system which can drive multiple measurement systems. Furthermore, we feel that policy languages are complex in nature and to maximize clarity, future iterations should become a completely independent, hand-crafted, language rather than a DSL embedded in C++, which for the purposes of this prototype was most readily applicable and prudent.

As discussed in chapter 6, a policy language like MSRR-PL can become an even more powerful tool for measurer development with the introduction of 'syntactic sugar‘ to decrease verbosity of policies and increase development velocity. Furthermore, partial generation tools which produce templates for measurer writers would also be advisable for an industry policy language.

Another avenue for the policy language would be to extend the validation functions and rule applicator functionality such that they support degrees of failure and degrees of passing. As prototyped, MSRR-PL supports rules that dichotomize into states into good and bad, or pass and fail. However, types of failures and degrees of severity will certainly prove useful for remote attestation in that it shall provide remote appraisers more information to make better informed decisions as to how to treat the target.

Autogeneration Techniques The difficult problem of automatically generating good measurement policies, of a similar nature and par to those written by a qualified expert, is likely to be the most challenging aspect of research into dynamic measurement, and the most rewarding.

In this paper we have explored the adaptation of symbolic execution techniques to the formation of behavioral expectations and finally measurement policies. We would like to see the leveraging of other techniques, both individually and in unison. We suspect that a system that produces policies via a host of static analysis approaches will function best to facilitate a strong ‘cocktail’ of measurement policy types. These automatic policies would form a strong base layer to potentially be augmented with developer written policies.

We suspect that there is much to be leveraged from the industry practices of unit testing, in regards to the structuring and formation of rule abstractions which evidence expected behavior. There is potential to capitalize further on the extensive work spent developing unit tests in current industry standard practices such as test driven development (TDD). It is probable that with minimal transformation, unit tests themselves may be selected to form the logic for measurement policies.

Moreover, simple techniques such as a ‘function mirroring’ approach could prove very scalable and very beneficial in regards to the autogeneration of measurement policies. The basic idea is that functions from the target application can be transformed, fairly easily and naively, into measurement policy validation functions. Using these mirroring validation functions, rules can be generated to sample the application at the original function’s entry and exit points to sample the live inputs and outputs of the function. The validation function itself will simply recompute the output from the live inputs to compare its calculation with the value computed live, in order to determine that the expected behavior is preserved.

Verifying Measurement System Integrity One critical consideration in remote attestation that we have left out of the scope of this work is the prospect that the measurements themselves must be verifiable to establish trust. This can be approached in various ways, ranging from proofs embedded in the protocol of the measurement itself to the technique of *meta-measurement*. In meta-measurement, an additional order of measurement service providers are utilized to measure and verify the measurers of the first order, like the MSRR Measurement System.

Chapter 9

Conclusions

In this work we have presented the MSRR Measurement Suite. The MSRR Measurement Suite is a system of tools that have been prototyped to demonstrate the ways in which the high cost of manual effort by qualified measurement experts can be reduced. To this end, we believe that we have been successful on several fronts.

We have prototyped a novel general purpose measurement system which contains the core measurement functionality common to all measurement scenarios. In this way, we have eliminated the need to build new measurement systems from the ground up for each new target application. This measurement system is lightweight and incurs a reasonable overhead on the target application.

We have employed a high-level, easy to use, first of its kind measurement policy language. This policy language allows one to describe expected behaviors for a target application and to specify sampling schedules to evidence such. Using this policy language and the measurement system, an expert can much more easily produce tailored measurement solutions at a per-application basis.

We have leveraged state of the art static analysis tools to generate policies automatically. The types of policies produced allow one to achieve, in the absence of an expert, a layer of high policy coverage for an entire system's application set, automatically. Furthermore, these generated policies can be complemented with the conserved and focused attention of an expert on critical application properties where concept-driven policy rules are desired.

In summary, we believe that the techniques we have prototyped with our suite are important steps in decreasing the cost of the synthesis of policies and tools to achieve reliable, efficient, and widespread adoption of remote attestation for user-space applications.

Bibliography

[1] Json.

[2] Algirdas Avižienis. Design of fault-tolerant computers. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 733–743, New York, NY, USA, 1967. ACM.

[3] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D&#x02019;elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[6] David Challener, Kent Yoder, and Ryan Catherman. *A Practical Guide to Trusted Computing*. IBM Press, 2008.

[7] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 144–155, New York, NY, USA, 2016. ACM.

[8] R Cieslak. Dynamic linker tricks: Using ld_preload to cheat, inject features and investigate programs. *Retrieved March*, 12:2015, 2015.

[9] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM.

[10] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O&#x2019;Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. *Int. J. Inf. Secur.*, 10(2):63–81, June 2011.

[11] George Coker, Joshua Guttman, Peter Loscocco, Justin Sheehy, and Brian Sniffen. *Attestation: Evidence and Trust*, pages 1–18. 2008.

[12] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *ACM Workshop on Scalable Trusted Computing*, pages 49–54, 2009.

[13] Michael J. Eager. Introduction to the DWARF debugging format. Eager Consulting at http://dwarfstd.org/, April 2012.

[14] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dy-

namic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007.

[15] Free Software Foundation. GDB: The GNU project debugger. https://www.gnu.org/software/gdb/, February 2018.

[16] John Gilmore and Stan Shebs. GDB internals. Cygnus Solutions, February 19 2004.

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[18] Trusted Computing Group. TCG infrastructure working group architecture part II: Integrity management. Specification Version 1.0, Revision 1.0, November 17 2006.

[19] Trusted Computing Group. TPM main part 1: Design principles. Specification Version 1.2, Revision 116, March 1 2011.

[20] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *Proceedings of Conference on Virtual Machine Research And Technology Symposium*, pages 3–15, 2004.

[21] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[22] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: Policy-reduced

integrity measurement architecture. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies*, pages 19–28, 2006.

[23] S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *Proceedings of the 17th Annual Computer Security Applications Conference*, pages 265–, 2001.

[24] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *2009 Conference on Dependable Systems Networks*, pages 115–124, June 2009.

[25] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[26] Ulrich Kühn, Marcel Selhorst, and Christian Stüble. Realizing property-based attestation and sealing with commonly available hard- and software. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, pages 50–57, 2007.

[27] Phillip A. Laplante. *What Every Engineer Should Know About Software Engineering (What Every Engineer Should Know).* CRC Press, Inc., Boca Raton, FL, USA, 2007.

[28] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, pages 21–29, 2007.

[29] K. Maeda. Performance evaluation of object serialization libraries in xml, json and binary formats. In *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, pages 177–182, May 2012.

[30] Andrew Martin. The ten page introduction to trusted computing. Technical Report RR-08-11, OUCL, December 2008.

[31] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[32] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.

[33] Milena Milenković, Aleksandar Milenković, and Emil Jovanov. Hardware support for code integrity in embedded processors. In *Proceedings of the Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 55–65, 2005.

[34] C. Mitchell and Institution of Electrical Engineers. *Trusted computing*. IEE professional applications of computing series. Institution of Electrical Engineers, 2005.

[35] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 605–615, New York, NY, USA, 2017. ACM.

[36] ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. Syminfer: Inferring program invariants using symbolic states. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 804–814, Piscataway, NJ, USA, 2017. IEEE Press.

[37] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Trans. Softw. Eng. Methodol.*, 23(4):30:1–30:30, September 2014.

[38] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[39] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 504–515, New York, NY, USA, 2011. ACM.

[40] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, pages 13–13, 2004.

[41] Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 103–115, 2007.

[42] Jonathan A. Poritz. Trust[ed , in] computing, signed code and the heat death

of the internet. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1855–1859, 2006.

[43] Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[44] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 49–64, Berkeley, CA, USA, 2015. USENIX Association.

[45] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[46] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Sci. Comput. Program.*, 64(1):54–75, January 2007.

[47] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42(4):443–476, April 2007.

[48] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 266–273, New York, NY, USA, 2004. ACM.

[49] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In

Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13, pages 16–16, 2004.

[50] M. Schroeder. A practical guide to object-oriented metrics. *IT Professional*, 1(6):30–36, Nov 1999.

[51] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 419–423, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[52] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 272–282, May 2004.

[53] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. *SIGOPS Oper. Syst. Rev.*, 39(5):1–16, October 2005.

[54] E. Shi, A. Perrig, and L. Van Doorn. BIND: a fine-grained attestation service for secure distributed systems. In *Symposium on Security and Privacy*, pages 154–168, May 2005.

[55] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2):10:1–10:34, May 2008.

[56] Open Source. Bluffinmuffin. `https://github.com/BluffinMuffin`.

[57] Open Source. Dreamchess. https://www.dreamchess.org/.

[58] Mark Thober, J. Aaron Pendergrass, and Andrew D. Jurik. Jmf: Java measurement framework: Language-supported runtime integrity measurement. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, pages 21–32, 2012.

[59] Mark Thober, J. Aaron Pendergrass, and Andrew D. Jurik. Ensuring the integrity of running java programs. 2013.

[60] A.H. Watson, D.R. Wallace, T.J. McCabe, McCabe & Associates, National Institute of Standards, and Technology (U.S.). *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Number v. 13 in NIST special publication. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996.