MSRR: Measurement Framework For Remote Attestation

Jason Gevargizian
EECS, University of Kansas
Email: jgevargi@ittc.ku.edu

Prasad A. Kulkarni
EECS, University of Kansas
Email: prasadk@ku.edu

***Abstract*—*Measurers* are critical to a remote attestation (RA) system to verify the integrity of a remote untrusted host. Run-time 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 (*appraiser*). However, existing run-time 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 run-time measurement. We describe *MSRR* (MeaSeReR), a novel general-purpose measurement framework that is agnostic of the target application. We show how measurement policies written per application can use MSRR, eliminating much time and effort spent on reproducing core measurement functionality. We describe MSRR's robust querying language, which allows the appraiser to accurately specify the what, when, and how to measure. We evaluate MSRR's overhead and demonstrate its functionality by employing real-world case studies.

I. Introduction

Remote attestation (RA) provides mechanisms for untrusted clients to prove their integrity to a remote party, and is integral for communicating entities to establish trust in a distributed computing environment [8]. In RA, an *appraiser* seeks to establish trust of a *target* by requesting *evidence*. Evidence is gathered by the target and returned to the appraiser; it can include static and configuration information, run-time measurements, and meta-evidence. A system is then said to be *trusted*, or *trustworthy*, from a point of reference, if it meets two criteria: (1) the system can be unambiguously identified, and (2) the system can be observed as behaving in accordance with previously known expectations [14].

Most remote attestation techniques provide integrity evidence by only measuring the *static* properties of the target hardware/software system [20]. Static software properties typically include the 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 run-time security attacks, such as buffer overflow attacks. *Static* RA techniques cannot measure the integrity of the software state that can dynamically change after the programs begin execution.

Dynamic RA attempts to remedy this limitation with static RA based systems [12], [13]. These dynamic RA approaches measure run-time properties of the executing software that must be true during a normal execution of the program, and are specific to each program. Measurement policies for dynamic RA 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. 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 system. Existing measurement systems are commonly built with their target application in mind, and which cannot fit any other software or possibly, even the different releases of the same application [13]. No general purpose measurement system with a well-defined and common command interface exists for native applications. We believe that the high cost of building customized measurement systems for user-level applications prohibits widespread adoption of dynamic RA in trusted computing.

While measurement policies for dynamic RA systems have to be application-specific, the measurement systems do not need to necessarily follow that trait. Our current work attempts to ease the process of building per application measurers by decoupling program-specific policies from the measurement system via a general, well-defined, and powerful command interface. We provide a novel general-purpose measurement framework, called MSRR, that can support different target applications. Our measurer provides the core measurement capabilities to sample programs features, such as the code, globals, heap, stack locals, and call-stack, as directed by application-specific measurement policies.

MSRR is implemented by extending the GNU Debugger, GDB [4], and currently resides as a trusted component on the same machine as the target application.

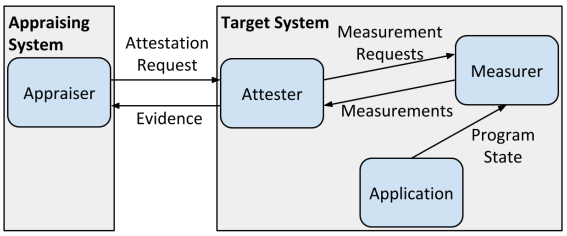

Fig. 1: Components of a remote attestation scenario

The measurer is driven by a remote attester that communicates with the measurer over JSON-RPC. This paper describes the context, features, implementation details, and evaluation of our novel measurement framework.

II. Background & Related Work

In this section we present the background for this work and other related approaches for remote attestation. RA is a mechanism for establishing trust. The key components to a typical remote attestation scenario are the **appraiser**, the **attester**, the **measurer**, and the (target) **application**, as shown in figure 1. The appraiser seeks to establish trust in a remote system by requesting evidence from the target by interfacing with the target's attester. The attester receives requests for evidence from remote appraisers and responds with evidence. To facilitate this task, the attester makes measurement requests of local measurers. The measurers collects measurements of trust critical features, including but not limited to user applications, as requested by the local attester.

a) Static Remote Attestation: To establish trust during RA, the appraiser needs to identify the remote system, and request and verify evidence regarding its hardware and software configuration [17], [6], [1]. The evidence supplied by many RA 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 [20]. Measured boot employs trusted hardware on the target machine, such as a TPM (*trusted platform module*) chip [7], 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.

SWATT [21] and Pioneer [22] 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 [16], [11].

b) Dynamic Remote Attestation: Static RA techniques cannot provide integrity evidence for the dynamic program properties and verify that applications are behaving *as expected* at run-time. Several works have studied run-time integrity measurement techniques. Flicker utilizes hardware support for late launch to bind and attest the integrity of executed code with its input/output [15]. BIND cryptographically attaches an integrity proof for a program with the output it produces [23]. 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 [10]. 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 program [8]. DynIMA combines load-time integrity measurement with dynamic tracking techniques that instrument program code to perform integrity-related runtime checks [2]. DynIMA only support dynamic checks that are generic to all binaries and require no program specific knowledge. No measurement interface is exposed.

Redas provides dynamic RA by measuring certain structural invariants and global data invariants [12]. Redas employs OS 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) defines a policy language to manually write program-specific run-time integrity policies, along with a measurement framework to sample the dynamic program state as directed by the security policies [24]. However, JMF only targets the Java environment and does not define a comprehensive measurement interface.

Several systems measure the Linux kernel runtime integrity. Copilot monitors certain well-known regions of Linux kernel memory using an external PCI card [18]. Another technique dynamically attests the Linux kernel control-flow integrity [19]. Linux kernel integrity monitoring, or LKIM, verifies the consistency of known critical kernel data structures at run-time [13]. These techniques develop custom measurement frameworks that only apply to the Linux kernel.

The cost of designing specialized measurement systems is high. While such costs can be amortized for OS kernels with their large code bases, they become prohibitive for smaller user-space applications. Our MSRR

framework presented in this work is unique because it is the first attempt to develop a comprehensive, extensible and well-defined command interface to decouple integrity measurement from generic policy specification for user-level binary applications.

III. Framework Interface and Capabilities

MSRR is a novel lightweight measurement framework that is agnostic of the target application. MSRR provides the core functionality to sample the execution state of a process. The decoupling provided by our measurement framework will allow an expert to save time and effort by only producing a program-specific measurement policy to drive the general measurement framework, rather than building the entire measurement system from the ground up for each user-level application.

MSRR has a robust querying language that provides the interface to the attesters. The querying language allows the attester 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. In this section we first describe the general capabilities of our measurement framework in Section III-A. We then present the measurement types, interface, and implementation details in Sections III-B, III-C, and III-D respectively.

A. MSRR Capabilities

MSRR can be driven locally or remotely via JSON-RPC. In a typical remote attestation scenario, the measurer would communicate exclusively with a local or remote attester, which in turn serves remote appraisers.

The measurement interface exposes control and measurement functions. The attester can select and deselect target processes to measure. The attester can request measurements of various features of program state, including but not limited to: call stack dump, local and global variable values, heap structures, type information for structures, and register and memory values.

Measurement requests are made using a querying language that is robust and allows for simple atomic measurement, complex composite measurements, or measurements with internal logic and dependent nested measurements. As such, requests can take different forms: immediate measurements, delayed or scheduled measurements, recurring or periodic measurements, and triggered measurements. All measurements types can be prefixed with logic contingent upon the result of previous measurements to support conditional short-cutting to avoid unnecessary target interruption and computation.

The measurement system is invokable via remote procedure calls and, by design, cannot invoke/message the appraiser at will. As such, non-immediate measurements immediately return an acknowledgment of successful scheduling and or failure to do so. When the scheduled measurement takes place, the measurement system stores this value. At any point, the appraiser may make a request for the result of a previously scheduled measurement; in such a case, the return will be the measured value or a "measurement pending" flag.

B. Measurement Types

In this section we present a classification of the measurement types supported by MSRR. The two main categories of supported measurement types are: *on-demand measurements* versus *monitoring measurements*. On-demand measurements are those sampled immediately, at the time of request. The samples are taken and evidence is returned back to the attester immediately.

Monitoring measurements are those that can be executed either periodically or upon some trigger, typically a reoccurring one. In either case, the measurements are registered with the measurer immediately upon receipt of the request. The measurer then stores samples when the specified trigger fires. The stored measurements can be requested at any time by the attester.

C. MSRR Command Interface

In this section we describe our measurement system's application programming interface. Table I shows the categories of MSRR commands and lists some important functions in each category. The first column in Table I 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.

MSRR exposes a command interface that is classified into the following categories.

Admin and Setup: These commands allow the appraiser 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 `load/store` commands start a monitoring measurement.

Features: These commands specify the program property to sample. They are used as arguments to the *measurement* commands.

Snapshots: These commands create a snapshot of the target application process. Application snapshots are discussed in Section III-D, and are an implementation utility to efficiently conduct expensive measurement tasks.

Events and Hooks: These commands assist the creation of the monitoring measurements by setting up events and hooks to perform such measurements.

Control Functions: These commands provide a higher-level interface to create more sophisticated mea-

Function	Arguments	Return	Description
Admin & Setup			
detach	-	VOID	Detach measurer from target.
quit	-	VOID	Terminate measurer.
set_target	STRING pid	VOID	Attach measurer to a process by PID.
Measurement			
load	INT store_id	MEASUREMENT	Load a measurement from a *store*.
measure	FEATURE f	MEASUREMENT	Measure a specific feature of target.
store	INT store_id, MEASUREMENT m	VOID	Store a measurement into in a *store*.
Features			
callstack	-	FEATURE	Create a *feature* representing the callstack.
mem	STRING address, STRING format	FEATURE	Create a *feature* for a specific memory address to be interpreted with a specific format.
reg	STRING reg_name	FEATURE	Creates a *feature* for a specific register.
var	STRING var_name	FEATURE	Creates a *feature* for a specific target variable, by source ID.
Snapshots			
snap	-	INT	Create a snapshot of target and return a reference to the snapshot.
to_snap	INT snap_id, STRING expression	Anything	Evaluate an expression on the specified snapshot.
Events & Hooks			
delay	INT delay, INT repeat	EVENT	Create a timer event with a specified duration.
disable	INT hook_id	VOID	Disable a hook.
enable	INT hook_id	VOID	Enable a hook.
hook	EVENT e, EXPRESSION a	INT	Create a hook that evaluates an expression when an event occurs.
kill	INT hook_id	VOID	Kill a hook.
reach	STRING source_file, INT source_line, INT repeat	EVENT	Create an event that triggers upon target reaching a specified code location.
reach_func	STRING func_name, INT repeat	EVENT	Create an event that triggers upon target reaching a specific function.
Control Functions			
eq	INT left_hand, INT right_hand	INT	Evaluate the equivalence of the arguments.
if	INT condition, EXPRESSION expr_true, EXPRESSION expr_false	Anything	Evaluate one of two expressions depending upon some condition.
not	INT original	INT	Return the boolean complement of the inputl.
seq	EXPRESSION e1, e2 ...	VOID	Evaluate a sequence of expressions.

TABLE I: MSRR extensible function interface

surements that will only be triggered if/when certain program properties (themselves, determined through earlier measurements) or conditions exist.

The MSRR commands return values of type `void`, `int`, `string`, `feature`, `measurement`, `event`, and `expression`. The `void`, `int`, and `string` types function as expected. *Measurement* functions use the `feature` type to describe a feature in the target application to measure. The `measurement` type has two main components, the data component and a type descriptor. 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. The `event` type is used to create *event* functions that trigger measurements periodically or when certain conditions are met. The `expression` type describes a lazy expression, that is currently used by the `hook` function to invoke other MSRR commands when the corresponding event is triggered. We provide some examples of MSRR functions in Section V.

D. Implementation

We have implemented MSRR as an extension of the GNU Project Debugger, GDB [4], [5]. 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 [3]. As such, MSRR requires the DWARF debug symbols for measuring the target applications. 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. 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.

Our GDB-based MSRR measurement framework utilizes hardware features, OS interfaces, and program instrumentation to provide its measurement capabilities. For instance, MSRR's `reach` and `reach_func`

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, MSRR utilizes GDB's *syscall* feature to pause the program at system calls for measurements, which uses interfaces exposed by the OS. The overhead of code instrumentation in MSRR is limited because the instrumentation is only inserted during the measurement period, and does not need to permanently slow-down the entire program execution.

Measurements in MSRR employ two strategies. *Direct measurements* are the default technique that stall program execution while recording the desired program state. For measurements that are expected to require a significant processing overhead, MSRR provides a strategy called *snapshot measurements.*

The following is the typical workflow for a direct measurement. On receiving a measurement request, depending on its type, MSRR performs an immediate sampling or schedules a sampling event. To ensure coherency, the actual measurement interrupts and stalls the target application for the entire duration of the search and processing of all data requested.

Typical Measurement Workflow:

1) Measurement request is received from the attester.
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 attester.

The snapshot measurement strategy is reserved for heavy-weight measurements. This strategy uses 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. The actual measurement happens on the child snapshot. The original target application is allowed to continue after the *fork* command returns.

Snapshot and Release Workflow:

1) Measurement request is received from the attester.
2) The measurer interrupts the target application.
3) An entire snapshot of the target's state is taken.
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 attester.

Figure 2 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 anticipated cost of taking the snapshot. In Figure 2, the measurement system receives the following measurement requests: first, a measurement request for **M1** and later a measurement request for 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 by the attester by using the MSRR commands interface.

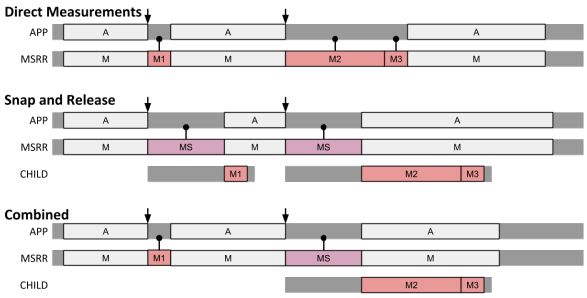

Fig. 2: Snapshot measurement scenarios

IV. Performance Benchmarking

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. In this section we present some performance benchmarking results with MSRR.

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 processors. We employ custom micro-benchmarks and programs from the SPEC CPU 2006 benchmark suite with the reference data sets [9]. Each benchmark-configuration is executed 10 times, and the average program run-time is used.

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

Our next 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

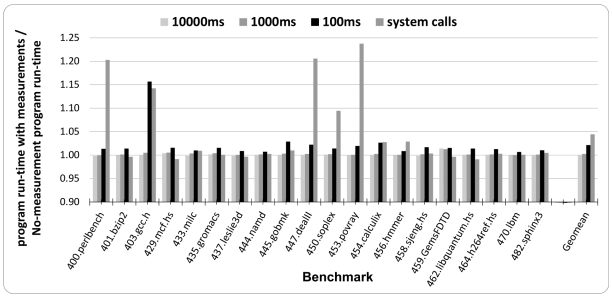

Fig. 3: MSRR overhead when invoked to periodically sample the application call-stack

program call-stack (`callstack`), a specific machine register (`reg`), and a stack memory variable (`mem`) every 10msec. 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 about 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 10,000msec.

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 `callstack`, `reg`, `mem`, `hook`, and `snap` events have an overhead of 0.54msec, 0.32msec, 0.32msec, 1.94msec, 96.45msec, respectively, on our system.

The cost of some events, especially `callstack` and `snap`, may vary depending on the client program's call-stack depth and memory usage. Our next experiment evaluates the overhead imposed by MSRR when sampling the entire call stack of a user-application at different measurement frequencies. Measurements are collected at periodic intervals of 100ms, 1000ms, 10,000ms, and at every system call.

Figure 3 shows the ratio of the program run-time when measurements are taken for various configurations to the program run-time with no measurer attached. We found that the measurer imposes an overhead of 0.08%, 0.25%, 2.14%, and 4.44% for call-stack measurements taken every 10,000ms, 1000ms, 100ms, 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 0.34% 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 (100ms, 1,000ms, or 10,000ms) for all benchmarks. For these timer-based experiments, the largest overhead was on benchmark 403.gcc at a measurement period of 100ms. 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.

V. Examples and Case Study

In this section we present a few examples to illustrate the MSRR function interface and invocation. Each example shows a simple exchange between an attester and our measurer. For each step in the exchange, we show a request made by the attester, followed by the response from the measurer. While the communication takes place using JSON-RPC, we only show the more readable short-form notation.

a) Attach measurer to process with PID 1590: This example shows the attester requesting MSRR to initiate measurement for a target application.

Command	(set_target 1590)
Response	INT:0

b) Sample the call stack immediately: Our second example uses the MSRR interface to request an on-demand measurement to sample the target application's call-stack.

Command	(measure (callstack))
Response	MEASUREMENT:M{type=0, cg=(main (sleep (__nanosleep_nocancel))), ft=1:main 2:sleep 3:__nanosleep_nocancel, next=NULL}

c) Store a measurement of variable A in function main: Our next example requests a *monitoring* measurement to sample the value of a variable on the application call-stack. The attester creates a hook which requires an event descriptor (**EVENT**) and an action (**EXPRESSION**). In this case the event is the *reach* function with an offset of 0. The action is a lazy expression that will sample and store the value in variable "A", when the event condition is satisfied. Since this measurement may not happen immediately, MSRR returns a '0' to simply indicate success.

Command	(hook (reach “main” 0 0) ‘(store 1 (measure (var “A”))))
Response	INT:0

d) Request the measurement stored in Store 1.: In this example, the attester requests MSRR to communicate the measurement stored by a previously registered hook. MSRR immediately responds with either the stored measurement, if available, else returns ‘0’.

Command	(load 1)
Response	MEASUREMENT:M{type=1, data=33 , next=NULL}

e) DreamChess case study: Next we present a longer realistic usage scenario for MSRR. We discuss some malicious modifications that an attacker may wish to perform on the application, and then we describe how our system may be used to detect such attacks. We employ *DreamChess*, which is an open source chess game for Windows, Mac, and Linux. The trust framework is tasked with ensuring that the game of chess is being played correctly and fairly. Using DreamChess, we describe a few high-level attestation goals. For each goal, we present a practical attestation policy generated by an expert. We also show how each policy translates into measurement requests in MSRR.

DreamChess describes the game board in a structure called *board*. The structure contains an integer array *square* of size 64, to represent the squares of the board. The first eight elements of the array correspond to the first row of spaces on the board. The next 8 elements correspond to the next row, 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 the following listing for the flags and the definition of *board*.

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

Goal – Is the Chess Board in a Valid State: The referee/appraiser may wish to determine if the board is valid, at any given time. The validity of a chess board can be defined in many ways. In this example we say a board is ‘valid’ if there are no more than 8 pawns of black and no more than 8 pawns of white.[1] As such, the rule may look something like the following.

[1]A complete valid board rule would be more complex to express all the rules of a chess game. It may bind the piece types and colors considering special chess piece promotion rules. It may also bind each piece to a subset of spots that it can reach.

```
(Status, Reason) ValidMove(int board[]) {
  if (board.countif(BLACK_PAWN) > 8)
    return (FAIL, "Too many black pawns!")
    ;
  else if (board.countif(WHITE_PAWN) > 8)
    return (FAIL, "Too many white pawns!")
    ;
  else
    return (PASS, null);
}
```

Listing 1: Valid Board Rule

For this rule, the appraiser may request a measurement of the square array within DreamChess’s board structure instance at any time. A measurement request and response for this rule would look as follows.

Command	(measure (board))
Response	MEASUREMENT:M{type=1, data={ 6,2,4,8,10,4,2,6, [. . .] 7,3,5,9,11,5,3,7 } , next=NULL}

Note that such a rule may not be true at program points when this structure is being legally updated. The rule could account for such legal rule violation by also requesting the measurer to return the *program counter* when the measurement is taken. The appraiser can then assess the measurement with the provided context.

Goal – Is the Move Valid An appraiser might also be interested in ensuring that each and every move made during play is valid. To do so, 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 the move was valid.

For this example, a move is defined as valid if the previous board and next board state differ in exactly two spaces. Again, this rule is simplistic since it does not account for complex maneuvers, such as *castling*, and does not bind each piece type to their specific move patterns. The simple rule can be described as follows.

```
(Status, Reason) ValidBoard(int board1[],
    int board2[]) {
  int diff_board[] = board2.subtract(
    board1);
  if (diff_board[].countNonZero()>2)
    return (FAIL, "More than two spaces
    changed!");
  else
    return (PASS, null);
}
```

Listing 2: Simple Valid Move Rule

For this rule, our measurement framework must be invoked as follows. First an initial (on-demand) measurement of the board is taken. Then, a recurring (monitoring) measurement is registered for each *move* operation. The appraiser can request the stored measurements at any time. The command to register the monitoring measurement is below.

Command	(hook (reach "DoMove" 0 1) (store 1 (measure (var "board->squares"))))
Response	INT:0

The appraiser can request the samples using the following rule.

Command	(load 1)
Response	MEASUREMENT:M{type=1, data={ 6,2,4,8,10,4,2,6, [. . .] 7,3,5,9,11,5,3,7 } , next=NULL}

VI. Future Work

There are multiple avenues for future work. First, insightful policy specification often requires manual input from an expert. We plan to explore static and dynamic techniques to automatically derive important program properties for policy specification during dynamic RA. Second, while MSRR provides a general-purpose command interface for driving measurements, the mapping from high-level policies to the actual MSRR commands is still manual. We plan to automate this mapping in future work. Finally, we will continue to extend the MSRR interface, improve its performance, reduce the trusted code base, ensure better separation between the target application and measurer, and develop protocols to attest the trustworthiness of the measurement framework and responses delivered to the remote attester.

VII. Conclusions

The goal of this work is to decouple application-specific measurement policies from the actual measurers that conduct the measurement at run-time in the context of dynamic remote attestation. We described our measurement framework, *MSRR*, that provides a general-purpose, comprehensive and extensible interface of commands to easily specify and efficiently perform dynamic program measurements. We presented MSRR's command interface, described its implementation details, evaluated its performance for a few measurement tasks, and illustrated its usage through multiple simple examples and case studies. In summary, we believe that our general-purpose measurement framework is an important first step in realizing the goal of automating, as much as possible, the synthesis of policies and tools to achieve reliable, efficient, and widespread adoption of dynamic RA for user-space applications.

References

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

[2] 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.

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

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

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

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.