A Logic for Secure Stratified Systems and its Application to Containerized Systems

Hagen Lauer*†, Amin Sakzad*, Carsten Rudolph*, Surya Nepal†
Monash University* Email: first.last@monash.edu
CSIRO's Data61† Email: first.last@data61.csiro.au

***Abstract*—We present the design and verification of a secure integrity measurement system for containerized systems. Containerization of applications allows fine-graded deployment and management of services and dependencies but also *needs* fine-graded security mechanisms. In this paper we provide formal abstractions for containerized systems by introducing LS^3, a formal model and logic with sub-domain constructs to represent stratified systems and their interactions. Using our formal model, we prove that the widely used Trusted Computing Group (TCG) based Integrity Measurement Architecture (IMA) securely extends trust measurements from boot to applications. However, IMA is not designed to make domain specific trust measurements and is consequently incapable of creating domain specific integrity reports. Current research aims to improve either trust measurement performance or comprehensiveness but does not improve the measurement function and its semantics to allow remote verification of measurements per domain. We present an enhanced trust measurement architecture design, which produces domain specific integrity measurements suitable for fine-graded remote attestation. Providing domain specific integrity reports eases system and sub-system verification and yields desirable properties such as measurement log stability and constrained disclosure for multi-domain systems. We verify and prove the correctness of our trust measurement architecture using our formal model.**

I. Introduction

Containerization is gaining increased popularity in a variety of settings and may be a key step towards future cloud computing models [8]. Containerization of applications removes the need to customize and configure an entire operating system in order to run a set of applications with their dependencies. Instead, an operating system is modified to include a *container-engine*, which runs and isolates containerized applications in their *virtual user-space* or virtual domain (Fig. 1). This concept is also referred to as *light-weight* operating system level virtualization because containers still share an operating system. Removing the overhead of running an entire operating system lets a customer handle her applications while a provider can handle everything else: instance selection, scaling, deployment, fault tolerance, monitoring, logging, security patches, and so on [8].

Since containerized applications share an operating system kernel, *trust* in a containerized application depends as much on the *integrity* of the host system as it depends on the hosted application itself. Only by inferring the integrity of the operating system *first*, we can infer the integrity of applications confined to their virtual user-spaces. For example, Alice may offer to host containerized applications and has set up infrastructure similar to Figure 1. Carol and Mallory both give a container with applications to Alice, which promises to run. As customers, Carol and Mallory *trust* Alice's setup to provide security properties: most importantly, isolation both in the sense of performance and protection. Being able to establish trust in the operating system and infrastructure becomes crucial if the provided service itself needs to be trusted. Alice needs a way of monitoring her host system for compromises. In addition, Carol and Mallory want to monitor and establish trust in the host system and the integrity of their container environments themselves. However, when reporting on the integrity of the system, Alice has to make sure that she provides *complete information* that is *constrained* to each container or otherwise she might lose trust or worse, give Mallory an incentive to launch targeted attacks against Carols applications.

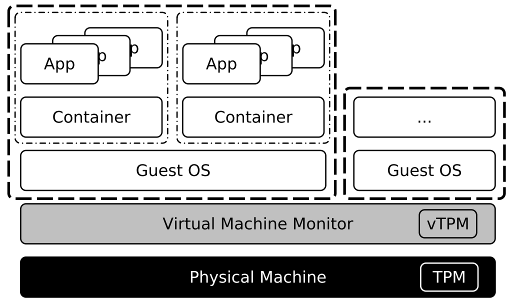

Figure 1. A container system running on top of virtualized infrastructure. The dashed line indicates the system components we are interested in: the operating system *domain* and contained applications in *sub-domains*. The operating system can run either bare-metal on the physical platform and Trusted Platform Module (TPM) or as a *guest* on a virtual machine monitor using a virtual TPM.

Container systems are specifically designed to not only give the illusion of a isolated user-space per container but to enforce it. Establishing its integrity is an important step towards establishing trust in applications running in their virtual environments. One way to establish trust in an operating system is to use the Trusted Platform Module (TPM) to record its software state. Using a process called remote attestation this information can be securely conveyed to a remote party. Given the software state, an agent may then infer the operating systems integrity and further properties to decide whether it can be trusted or not. While the TPM is capable of securely storing and reporting software recordings, a *measurement function* within the target system is required to take and extend measurements to the TPM in a trustworthy manner. Trusted and secure boot [5] create a chain of trust measurements from system boot all the way to when control is transferred to an operating system kernel. The Integrity Measurement Architecture (IMA) [15] is a kernel function that immediately continues chain of trust measurements by recording code and data before it is mapped to the system memory.

The existing body of research on trust measurements is focused on *comprehensiveness* and *performance*. Examples include moving from measuring code at load-time to measuring memory segments during run-time [14] on one side and policy reduced measurement targets [16], [18], e.g. kernel modules only, or batch-extend techniques on the other side so as to increase the overall performance of the measurement function. None of these target measurement semantics, remote verifiability, and constraining disclosure. Formal investigations targeting secure execution environments [19] are complementary and can be combined with our work.

We propose a measurement architecture with constrained disclosure features built in by making domain specific measurements to begin with. We argue that we preserve comprehensiveness and other desirable properties. To this end, we use the modular design of *LS*^{2} [3] and add to it domains. Domains are logical *strata*, i.e. VMM, OS, container, with a logical relation defining control and interface confinement. In short, the operating system and its threads are the primary domain. We then create *lightweight* domains under the control of but with interfaces to a primary domain. This corresponds to an operating system controlling and managing all memory while providing each container with a partition of *virtual* addresses. Using the idea that resources are allocated to domains, we simplify a proof made with *LS*^{2} in our extension titled *LS*^{3}. We extend prior work to prove the trustworthiness of measurements taken by IMA. We show that through our domain-subdomain construct measurements can be recorded into dedicated TPM locations and we link them with a technique called *forward linking*. Consequently, when we wish to report on the software state of a container and its host operating system, we can report on a domain and a particular sub-domain in a fine-graded manner instead of divulging all measurements. We prove the properties with the same rigor as *LS*^{2} has proven a sequential boot sequence and show that while we still measure *everything*, through sorting and linking we achieve a much more verifiable and constrained system state per container.

Our contributions are:

- LS^3 extensions for containerized systems in Section III,
- the design and verification of our Enhanced Integrity Measurement Architecture (EIMA) in Sections III and III-C,
- a proof of the trustworthiness of integrity measurements for IMA in addition to constrained disclosure in EIMA (Section III-C).

II. Background and Prior Work

A. *LS*^{2}: A Logic for Secure Systems

Our paper draws from the successful analysis performed in [3] which presented a logic for secure systems or *LS*^{2}. We build upon *LS*^{2} and use what [3] refers to as constrained by system interface (CSI) adversaries, which are simply modeled as extra threads existing on the target system. Our extension exploits LS^2's modularity and allows us to *explicitly* create new threads executing untrusted code. We found that since LS^2 was intended for analyzing only a small part of trusted computing based systems relying on *loading* and *jumping* to new code, reasoning about *launching* a complex system at kernel and user level threads are impossible. We extend the capabilities of LS^2 well into application space and potentially virtual machines to fulfill some of the promises made in [3].

LS^2 is a metalanguage which has two main design goals: (i) allow for the abstraction of thread based sequential execution of programs, and (ii) provide a proof system and logic for defining and verifying desirable *safety* and *security* properties. We revisit LS^2 [3] as a three part formal system consisting of an impure functional programming language, reduction-based operational semantics, and a first-order logic system with hybrid extensions over security properties. The programming environment has a syntax close to that of functional programming languages like ML [13]. Most notably, programs are a sequence of *actions*, represented as a stack in [7] and need only the sequential composition operator ";" to indicate that one action is to be executed after the other. The set of *actions* conveniently abstract manipulation of memory locations, network communication, anonymous function definition and evaluation, as well as simple program flow controls based on comparing values instead of *if-else* constructs. Most notably, LS^2 has explicit actions for obtaining and releasing write locks on (memory) locations. Such programs end with $\cdot$ or "no-op" or with a *jump* to another program:

$$\begin{array}{llll}
\textit{Location} & l & & \\
\textit{Expression} & e & ::= & n \mid x \mid (e, e') \mid \ldots \\
\textit{Actions} & a & ::= & \texttt{read } l \mid \texttt{write } l, e \mid \texttt{hash } e \mid \\
 & & & \texttt{lock } l \mid \texttt{eval } f, e \mid \texttt{match } e, e' \mid \\
 & & & \ldots \\
\textit{Program} & P & ::= & \cdot \mid \texttt{jump } P \mid x := a; P
\end{array}$$

The accompanying parallel composition operator "|" is used to execute entire programs in parallel using *threads* which supports concurrent execution by *interleaving*. The companion machine model of LS^2 can be summarized as follows:

$$\begin{array}{llll}
\textit{Thread ID} & I & & \\
\textit{Thread} & T & ::= & [P]_I \\
\textit{Store} & \sigma & : & l \mapsto e \\
\textit{Lock Map} & \imath & : & l \mapsto I \cup \{\_\} \\
\textit{Configuration} & C & ::= & \iota, \sigma, T_1 | \ldots | T_n
\end{array}$$

Threads are identified using I, which may be a triple identifying *owner, machine,* and a *name* that we have not included above. All executing threads I sequentially reduce actions of a program $[P]_I$. The systems memory is modeled using σ which maps locations l of machines to expressions e, read as "l holds e". Most notably, *LS*^{2} uses a global lock map which allows to specify whether a particular memory location is *unlocked* or *locked* by a thread I, i.e. modifiable only through a program or action $[P]$ reduced by I. Finally, configurations C include mappings of locations, locks, as well as running threads.

LS^2 uses explicit *operational semantics* to explain *how* one configuration transitions to another. Each transition has a *rule* in the operational semantics detailing its effect as the following example shows:

$$\begin{array}{l}
[\texttt{jump}\ P]_I \rightarrow [P]_I \\
[x := \texttt{hash}\ e; P]_I \rightarrow [P(Hash(e)/x)]_I \\
\iota[l \mapsto \_], [x := \texttt{lock}\ l; P]_I \rightarrow \iota[l \mapsto I], [P(0/x)]_[I] \\
\iota, \sigma[l \mapsto e'], [x := \texttt{write}\ l, e; P]_I \rightarrow \iota, \sigma[l \mapsto e], [P(l/x)]_I
\end{array}$$

Transitions from $C \rightarrow C'$ are written with the *redex*, i.e. the state to be changed, on the left hand side of the arrow pointing to the *reactum*, i.e. the changed state, on the right hand side. For example, when $\texttt{jump}\ P$, $x := \texttt{hash}\ e; P$, $x := \texttt{lock}\ l; P$, and $\texttt{x} := \texttt{write}\ l, e; P$ are reduced by thread I. As expected, a $\texttt{jump}\ P$ has the effect that I executes actions of $[P]$ next. Reducing $x := \texttt{hash}\ e; P$ has the effect that I continues with $[P]$ while x is now bound to $Hash(e)$. Similarly, $\texttt{lock}\ l; P$ returns 0 and requires that l is not locked when it is reduced. The effect states consequence that l then maps to I and I continues with $[P]$.
Each transition is then labeled with a covariant index t such that transitions can be labeled $C \xrightarrow{t_0} C' \xrightarrow{t_1} C'' \xrightarrow{\cdots} C \ldots$. The labeled transitions in combination with the new state are referred to as the *trace* of the system and t may be interpreted as *time*. We can then reason about configurations which must have happened *before* another. Lastly, Datta et al. construct a predicate logic with temporal extensions. Formulas of the logic support the usual connectives and between *general* and *action* predicates.

Action	R	::=	$\mathsf{Read(I, l, e)} \mid \mathsf{Write(I, l, exp)} \mid \mathsf{Hash(I, exp)} \mid \ldots$
General	M	::=	$\mathsf{Jump(I, P)} \mid \mathsf{Mem(l, e)} \mid \mathsf{Locked(I, l)} \mid e = e' \mid t \geq t' \mid \ldots$
Formulas	A, B	::=	$t \mid \imath \mid \top \mid \bot \mid R \mid M \mid A \wedge B \mid A \vee B \mid A \supset B \mid \neg A \mid \forall x.A \mid \exists x.A \mid A\ @\ t$
Modal	J	::=	$[P]_I^{t_b, t_e} A \mid [a]_{I,x}^{t_b, t_e} A$

Actions reduced by threads I are presented as action predicates and are kept separate from general facts and machine specific predicates, i.e. such as equality of expressions, memory states or locks of a location. The formula $A\ @\ t$ is used to capture "A is true at t" [2], e.g. $\mathsf{Jump(I, P)}\ @\ t$ is true if I reduces $[\texttt{jump}\ P]$ at t. LS^2 often uses intervals over traces in the usual form using (t_1, t_2),$[t_1, t_2]$,$(t_1, t_2]$, and $[t_1, t_2)$. For intervals i, A on i is defined as A holds on all points t in i. Security and safety properties are expressed as one of two modal formulas. Formula $[P]_I^{t_b, t_e} A$ means that formula A is holds or is true whenever thread I executes P in the right-open interval $(t_b, t_a]$. A can be used to express security properties of P and may contain variables unbound in P.

B. Trusted Computing

The Trusted Platform Module (TPM) is a component of the physical machine in Figure 1 and implements the specification defined by the Trusted Computing Group [6]. From an abstract perspective, a TPM can be described as an extra hardware chip equipped with a public and private key pair $\{K_{TPM}, K_{TPM}^{-1}\}$ and a set of secure memory regions known as Platform Configuration Registers (PCRs) (details in subsection II-C). The TPM specification requires that the private key K_{TPM}^{-1} to be kept secret and the PCRs be resistant at least against software attacks. The private key may be used to sign contents of PCRs, and if an external verifier knows K_{TPM} she can verify that the signed value is indeed a PCR value. The process of signing PCRs on a target platform and evaluation of PCRs is part of the process called remote attestation. In fact, a verifier assuming malicious software on the machine can only trust the signed PCR values, i.e. $K_{TPM}^{-1}(PCRs)$. Hence, we need to convey facts about the target system using PCRs only and use them to record *configurations* of a platform in a trustworthy manner.

C. Platform Configuration Registers (PCR)

PCRs are treated as a special form of memory with only one dedicated command to *modify* them: $\texttt{extend}\ l, e$. Our $\texttt{extend}$ corresponds to the command *TPM_extend* of [6] and requires that l be a PCR index or handle, and e be a hash of e. Each PCR represents a sequence $\langle sinit, v_1, \ldots, v_\nu \rangle$ with *sinit* being its initial value to which values $v_1, \ldots, v_\nu$ must have been added *sequentially* using $\texttt{extend}$ for each value v. The only way to reset a PCR is to reboot the physical machine for which *LS*^{2} has a dedicated rule and general predicate. A TPM singed sequence $\langle sinit, v_1, \ldots, v_\nu \rangle$ lets a verifier gather that (a) the machine hasn't been restarted and (b) that $\texttt{extend}\ l, v_1$ must have happened *before* $\texttt{extend}\ l, v_2$ and so forth. PCRs and the key pair $\{K_{TPM}, K_{TPM}^{-1}\}$ form the root of trust for storage (RTS) and reporting (RTR) of a physical machine [6].

D. Root of Trust for Measurement

The RTS and RTR are completed by a root of trust for measurement (RTM) [6]. Informally, the RTM is a thread I on the physical machine which reduces or executes $\texttt{extend}$ / *TPM_extend*. The root of trust for measurement is responsible for initiating a chain of trust measurements. When a machine is powered on, the first program executed is the *core root of trust for measurement* (CRTM). The CRTM is responsible for measuring, i.e. hash and extend, the first piece of code in a boot sequence, e.g. the BIOS. The CRTM program itself can not be measured and a verifier has to trust that a machine executes exactly the sequence CRTM whenever the physical machine is booted.

E. Measured Boot

The process of creating a chain of measurements from CRTM to the operating system is referred to as Measured

Boot (Fig. 2). The chain of trust measurements initiated by the CRTM is the continued using the following program pattern:

$$[p := \texttt{read}\ P; h := \texttt{hash}\ p; \texttt{extend}\ h; \texttt{jump}\ p] \qquad (1)$$

We have omitted known locations l. This sequence is started by the CRTM using the BIOS code as P, and continued by the BIOS. The BIOS code reads boot loader (BL(m)) code and jumps to it, and finally, the BL(m) code reads operating system (OS(m)) code and jumps to it. In the PCR used by CRTM(m), BIOS(m), and BL(m), this produces the following sequence:

$$\langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m)\rangle \qquad (2)$$

This patten in Equation 1 can have an arbitrary number of programs in between as long as the pattern of reading some P, extending the hash of P, and jumping only to P is maintained.

F. Integrity Measurement Architecture (IMA) [15]

The TCG-based Integrity Measurement Architecture (IMA) proposed in 2004 [15] has since been adopted in the security sub-system of the Linux kernel (version $\geq$ 2.6). Today, IMA forms the basis of many integrity verification frameworks which aim for run-time integrity verification [14], policy-based attestation [16], and is actively being used, studied, and adapted to increase overall measurement performance [11], [12], [18]. The sole purpose of IMA is to extend the chain of trust measurements from the OS well into *user-space* by measuring potentially *everything* that is mapped to system memory beyond the OS, including loadable kernel modules, applications, and configuration files. For practical reasons, we will focus on OS components such as kernel modules and *user-space* applications. IMA is part of the operating system and once the OS is running, the sequence in equation 2 in the PCR used to boot the OS is hashed and extended into a fresh PCR for IMA. This creates the sequence $\langle sinit, \langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m)\rangle\rangle$ in the PCR used by IMA. By *hooking* onto loader functionality, i.e. the part of the OS responsible for memory mapping, IMA is able to continue the pattern of reading, hashing, and extending values into a PCR. For example, launching an application creates results in the sequence $\langle sinit, \langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m)\rangle, \text{App}(m)\rangle$ in the PCR used by IMA. Effectively, IMA continues a Measured Boot by performing a measured launch of *user* applications.

G. Security Properties

The following security properties have guided our formal analysis throughout this paper:

1) The *Integrity of a System* (such as the operating system running on a machine) is not compromised if no unauthorized modification to the system configuration can be made without the modification being recorded. We limit the scope of modifications to loading kernel modules and launch applications, i.e. events we can measure and record.

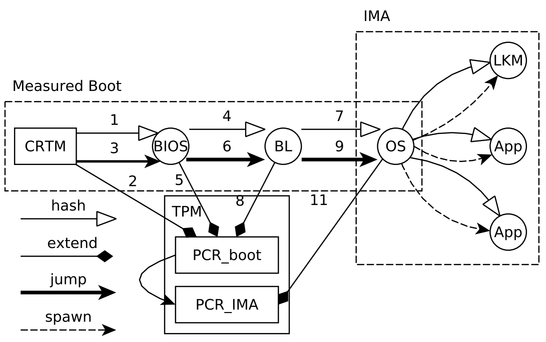

Figure 2. The `read`, `hash`, `extend`, then `jump` sequence of Measured Boot ends at but includes the operating system. Instead of switching the program of $[\text{OS}(m)]_I$, I spawns new threads running kernel tasks [LKM(m)] and *user-space* applications [App(m)].

2) *Trustworthiness of Measurement* is maintained if the measurements taken reflect the system configuration with regards to its *integrity*. A version of this property forms the basis of the formal analysis in [3] but is not extended beyond loading operating system code.
3) *Forward Integrity* of the measurement log is achieved if during a *run*, i.e. between reboots, recorded events can not be deleted, removed, hidden, or overwritten. *Forward Integrity* is the main property of IMA and the reason why PCRs are used to record events.
4) *Adversary Confinement* is achieved when code supplied by an adversary is recorded before its actions can take effect. Although not explicitly mentioned, both IMA and Measured Boot inherently provide this property by recording read code into a PCR before further actions can be reduced.
5) Lastly, *Constrained Disclosure* is a desirable property of any recoding and reporting procedure. While recording any modifications of a system, we report only the modifications which can affect a systems *integrity*. For instance, when reporting on the operating systems integrity, we don't need to include reports of *user-space* applications. Similarly, when reporting the integrity of a sub-domain, we need a report of the host domain but not of other sub-domains of the system.

In this work, we focus on security property (2) and (5). Security properties (1), (3), and (4) can be achieved using properties of the TPM and a measured boot sequence [17].

III. *LS*^{3} AND DOMAIN SPECIFIC MEASUREMENTS

Separation between components and the *Principle of Least Authority* (POLA) are key ingredients of secure systems [4] and compartmentalization is generally regarded as a good engineering practice [9]. Typically, modern computer systems have different abstraction and security layers with different privileges such that components in the same layer may not interfere with each other without the supervision of a lower layer and higher privileged component. We refer to this

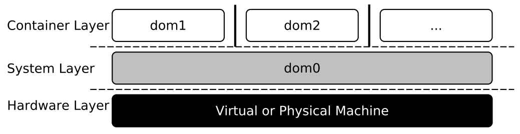

Figure 3. *LS*^{3} *domains* and *sub-domains* in relation to the machine which may be physical or virtual. Dashed lines indicate stratification using different *layers* and solid lines between *sub-domains* indicate compartmentalization using containers.

as system-*stratification* and component-*compartmentalization* (Fig. 3). Containers as a summarizing concept for operating system level virtualization techniques allow us to create *virtual user-spaces* to run applications directly on the host operating system, with restricted and possibly minimal access to shared resources. Containers are compartments on the user-level. Consequently, the host operating system controlling *all* system resources will be represented as a *domain*. Containers, dependent on a host operating system, are referred to as a *sub-domain*.

In our review of *LS*^{2}, we have described its system model using *Machines*, *Threads*, *Storage*, and *Locks*. Threads in *LS*^{2} are associated to machines using an identifier structure I which contains a reference to a machine identifier m. In order to reason over which thread must have caused an effect on a storage location such as PCRs, *LS*^{2} uses locks as a reasoning tool. Locks are necessary in *LS*^{2} as there is no stratification and no privilege separation, i.e. all threads, including those executing adversarial code can access any data structure unless it is explicitly locked.

LS^3 adds *domains* and *sub-domains*. First, we associate threads with a domain and for now, there is only $dom0$ and dom_n ($n > 0$). Dom_0 is a domain model borrowed from Xen [1] which includes the operating system kernel, kernel level threads, and possibly privileged user-space with management interfaces for a service provider. Containers, or *virtual* user-spaces, are sub-domains dom_n with $n > 0$. To reflect a threads association with a particular domain we extend the thread identifier I to include a domain identifier d. With n as its given name, m to indicate the machine, and d its domain or privilege level on m. Threads I in *LS*^{3} run programs *LS*^{2} programs P including all actions described in [3]. However, *LS*^{2} only has locks, and locking a location l for each thread I is not practical. Instead, we represent container semantics by partitioning our machine's storage according to the domains. We say that a machine m has storage of type $disk$, ram, or pcr. Each type of storage can be partitioned by adding a domain name or storage ID after it.

Thread ID	I	::=	(n, m, d)
Thread	T	::=	$[P]_I$
Machine	m		
Storage Type	sty	::=	disk | ram | pcr
Location	l	::=	$m\ .\ sty\ .\ ref$
Store	σ	:	$l \mapsto e$
Allocation	α	:	$l \mapsto d$
Configuration	C	::=	$\alpha, \sigma, T_1 \| \dots \| T_n$

For example, a *container-image* which holds a programs P to be executed in the corresponding container domain d_n ($n > 0$) is stored at location $l.disk.dom\_n.P$. In this case ref would be $dom\_n.P$ where dom_n is the partition in which we would look for P. Similarly, storage types ram and pcr can be partitioned. The partitioning is enforced in the operational semantics for `read` and *write*:

$$\alpha[l \mapsto I_d], \sigma[l \mapsto e'], [\texttt{write}\ l, e; P]_I \rightarrow \alpha, \sigma[l \mapsto e],\ \dots$$
$$\alpha[l \mapsto I_d], \sigma[l \mapsto e], [x := \texttt{read}\ l; P]_I \rightarrow \alpha, \sigma[l \mapsto e],\ \dots$$

The system property which states that domain dom_0 controls all resources can be expressed by adding the default mapping to our allocator:

$$\alpha : * \mapsto dom_0$$

Allowing multiple entries in α for a location l allow us to model logical sub-partitions for dom_n under the control of dom_0. This also gives us the basis for reasoning about *effects* in the system. For example, an expression written in a location allocated to some domain dom_n could only have been written by some thread in dom_0, i.e. the host system, or by a thread of dom_n but not by a thread of another untrusted domain $dom_m,\ m \neq 0, n$.

A. Measured Boot and IMA in LS^3

The first application of *LS*^{3} is the chain of trust measurements from system boot to application launch which we do in two steps: in step one, we discuss measured boot using our machine model and in step two, we extend the measurements by explicitly adding threads to our system.

To express measured boot, we need a predicate for a machine start, reboot, or reset. Following *LS*^{2} we introduce the *general predicate* $\mathsf{Reset}(m, I)$ which states that machine m is restarted producing the initial thread I. The corresponding rule in the operational semantics states that all volatile memory, i.e. RAM and PCRs, is reset and allocations on m are removed:

$$\begin{aligned}\mathcal{C} \ \longrightarrow\ & \alpha \setminus (m.l.*), \sigma[m.pcr.* \mapsto sinit] \setminus (m.l.ram.*), \\ & (T_1|...|T_n) - T_m \mid [\text{CRTM}(m)]_I)\end{aligned}$$

Upon a reset, all threads on m are removed and replaced by *just* one thread $I_{(n,m,dom_0)}$ executing exactly the program CRTM(m). The program CRTM(m) corresponds to the only piece of trusted code that is executed right at the beginning. When a machine in *LS*^{2} is reset, the same thread I is created but it is assumed that there may be more than this thread after $\mathsf{Reset}(m, I)@t$. Hence, in order to reason over which thread extended the first piece of code into a PCR, *LS*^{2} resorts to global locks. Threads in *LS*^{3} are created explicitly which brings us the following lemma:

Lemma 1 (Replacing locks by launching only I): When a machine is reset in *LS*^{3} at time t_R, only a thread I is created in the default domain dom_0. For any location l allocated to dom_0 in α at time t_R, we have

$$\forall t', t''.(t_R < t' \leq t''), \nexists e, \mathsf{Mem(l,e)}@t', \neg\mathsf{Write(l,e)}@t''.$$

The following program sequence (first shown in Equation 1) is used to bootstrap the operating system in Figure 2:

$$[p := \texttt{read } P; h := \texttt{hash } p; \texttt{extend } h; \texttt{jump } p] \qquad (3)$$

By replacing the generic sequence above with P as BIOS(m), BL(m), and OS(m) we can construct a chain of measurements from CRTM(m) to OS(m). That is, if all programs in the sequence are *exactly* those programs and if they are executed fully. Programs are typically loaded from the $disk$ of m which is generally regarded as untrusted, meaning that anything we read from the disk is potentially adversarial code. In order to establish that the sequence in the PCR used by P, i.e. PCR_boot in Figure 2, implies that the correct boot sequence has been executed, we need to reason step by step starting at CRTM(m). We get proof of the execution of CRTM(m) by finding the BIOS(m) logged in the PCR. However, by finding BIOS(m) in the PCR a verifier can only infer that action `extend` has been reduced of CRTM(m). In oder to establish that CRTM(m) has been *fully* and *exclusively* executed, i.e. no further code, we need to use the predicate $\mathsf{Jump(I, P)}$. Using $\mathsf{Jump(I, P)}$ as a tool for reasoning [3] lets us connect proof of the execution of BIOS(m) with a property we wish to prove about I executing CRTM(m). By finding BL(m) we get prove that CRTM(m) has not only measured and recorded BIOS(m) but must have also *jumped* to it. This lets us prove an important security property: *Trustworthiness of Measurement.*

This style of reasoning using the predicate $\mathsf{Jump(I, P)}$ can be continued infinitely but will always *exclude* the last element because we are not able to prove *full* or *exclusive* execution [3]. Hence, the sequence in Equation 1, without assumptions about the execution environment, lets us prove to a verifier that at least some portion of the OS(m) actions has been reduced.

$[\text{OS}(m)]_I$ represents known minimal operating system kernel running in dom_0 on m in control of all resources of that machine. It becomes critically important that any *runtime* additions to this kernel are recorded in order to provide or verify the systems integrity [15] (Section II-G). IMA is designed to fill this gap and extend the chain of trust measurements further into a running system. As mentioned earlier, to model this properly *LS*^{3} lets us add threads *explicitly* using the action `spawn` P along with the according operational semantics rule and general predicate $\mathsf{Spawn(I, P)}$

$$C, [\texttt{spawn } P;\ Q]_I \rightarrow \{T\} \cup T_J, [Q(\dots)]_I, [P]_J, \\ (dom(J) = dom(I))$$

Where we specify that the expression of action `spawn` is a program (e:P), the newly created thread J executes P, and that the caller I continues with the remainder of her program Q. As such, `spawn` corresponds to allocating executable memory for a thread in dom_0 containing P, mapping P to some $m.ram.J$, assigning J to the domain of the caller, and scheduling J for execution.

This construct allows us to continue reasoning over the actions of program OS(m) and how IMA continues the chain of trust measurements in a trustworthy manner. For convenience, we make the IMA kernel code itself measurable by defining the following programs for OS(m) and LM(m), i.e. the OS and the loader module (LM):

$$[pcr := \texttt{read } PCR\_boot; h := \texttt{hash } pcr; \\ \texttt{extend } h, PCR\_IMA;\ \texttt{spawn } l.ram.OS.LM]_{\text{OS}(m)}$$

and finally LM(m):

$$[p := \texttt{read } P; h := \texttt{hash } p; \\ \texttt{extend } h, PCR\_IMA;\ \texttt{spawn } P]_{\text{LM}(m);\dots}$$

The OS(m) is responsible for *linking* the PCR values used to boot the machine to the PCR which will be used to record loadable kernel modules (LKMs) and applications by the program LM(m). For simplicity, we say that LM(m) is a sub-program of OS(m) and that by measuring and recording the operating system, we also get a hash of LM(m). The PCR configuration of m *before* the LM(m) visible to a verifier is then:

$$\langle sinit, \text{BIOS(m)}, \text{BL(m)}, \text{OS(m)}\rangle_{PCR_{boot}}$$

$$\langle sinit, \langle sinit, \text{BIOS(m)}, \text{BL(m)}, \text{OS(m)}\rangle\rangle_{PCR_{ima}}$$

Based on this the verifier can only infer that OS(m) must have reduced the actions before [`spawn`], but not that the loader module is in fact running now. Once the first LKM or application has been loaded, the PCR used by IMA will change to:

$$\langle sinit, \langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m)\rangle, \text{P(m)}\rangle_{PCR_{ima}}$$

As before, P(m) could be any *untrusted* program on the disk, and may or may not allow us to reason about whether or not it is being executed. However, using the predicate and the rule $\mathsf{Spawn(I, P)}$ similar to a *jump* to link knowledge of the OS(m) and the fact that only some thread running LM(m) could have extended PCR_{ima} allows us to prove to a verifier that IMA is running and performing measurements.

B. LS^3 *Containers and Domain Specific Measurements*

Physical co-residency is the center of hardware-level side-channel attacks in the cloud [8], [10]. As a first step in these types of attacks, the adversarial tenant needs to confirm the cohabitation with the victim on the same physical host, instead of randomly attacking strangers [8]. For this reason, we argue that it is not acceptable to use the IMA code in the operating system loader component shared among domains.

In *LS*^{3}, containers are represented by *container-images* on a machines disk, and by a container-engine sub-system part of the operating system. The container engine is *trusted* to maintain isolation in the sense of performance and protection. In short, we assume that a container engine part of the operating system manages containers. Before a container is run, we assume that the store σ and allocation α have been prepared: both $m.disk$ and $m.ram$ receive a partition dom_n, $(n > 0)$. The disk region $m.disk.dom_n$ corresponds to the container image and the memory is used to execute programs shipped with the container. The allocator is prepared to enforce this accordingly by adding $m.ram.dom_n \rightarrow dom_n$ and $m.disk.dom_n \rightarrow dom_n$.

To allow for *spawning* threads running programs of the container we add an *overloaded* command spawn P, d to our system which adds a parameter for the target domain of the newly scheduled thread.

$$C, [\texttt{spawn}\ P, d;\ Q]_{I_{dom_0}} \longrightarrow \{T\} \cup T_J, [\ldots]_I, [P]_{J_d}$$

Most importantly, the command for spawning new processes in domains can only be reduced by a thread of dom_0, i.e. the host operating system. Modifying the loader code to use spawn P, d is intuitive but violates the *constrained disclosure* property: Instantiating LM(m) with spawn P, d would necessarily result in PCR_{ima} holding hash values of *all* programs ever spawned or loaded on m between a reboot. A remote verifier wishing to attest the system would learn not only about programs of her container but also about other programs which may currently be running on the system. Measuring *everything* into *one* record invalidates any efforts to randomize the distribution of services and acts like a system layout reverser for an attacker.

We propose an Enhanced Integrity Measurement Architecture (EIMA) which extends IMA measuring newly launched programs in a way which respects system isolation and partitioning. We say that EIMA produces domain specific measurements. When a new container is setup on our system we also partition PCRs. Before adding the allocation entry we *link* the PCR for a domain with the PCR of the host system:

$$\langle sinit,\ \langle ... \rangle_{PCR_{dom_n}} \rangle_{PCR_{dom_n}} \tag{4}$$

After that, we add the entry $m.pcr.dom_n \rightarrow dom_n$ to our allocator α. Bootstrapping the container environment after that can be done by spawning an *init*-like process from the container image.

$$\begin{aligned}&[p := \texttt{read}\ m.disk.dom_n.Int; h := \texttt{hash}\ p;\\ &\quad \texttt{extend}\ h, pcr_{dom_n};\ \texttt{spawn}\ P, dom_n]_{\text{ELM}(m);\ldots}\end{aligned}$$

The modifications to the *enhanced loader module* (ELM(m)) are minimal: we read a program Init$(\ldots)$ from the container image located at $m.disk.dom_n$, hash it and extend it to pcr_{dom_n} before spawning a new process running the program in dom_n. Figure 4 shows three domains: dom_0 as the domain created after a machine reset and dom_1, dom_2 as sub-domains or isolated *user-spaces* hosting different applications as per our introduction. Having a fully booted host system running containers as in Figure 4 produces the following PCR configurations:

$$\langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m) \rangle_{PCR_{boot}} \tag{5}$$

$$\langle sinit, \langle ... \rangle_{PCR_{boot}}, \text{LKM}(m),\ ... \rangle_{PCR_{dom_0}} \tag{6}$$

$$\langle sinit,\ \langle ... \rangle_{PCR_{dom_0}},\ \text{App1}(m),\ \text{App2}(m), ... \rangle_{PCR_{dom_1}} \tag{7}$$

$$\langle sinit,\ \langle ... \rangle_{PCR_{dom_0}},\ \text{App3}(m),\ \text{App4}(m), ... \rangle_{PCR_{dom_2}} \tag{8}$$

Hence, each container can be remotely verified using only *relevant* information, i.e. system boot, host operating system, and sub-domain measurements. Consequently, a service provider hosting containers can offer *trustworthy measurements* for containerized environments without having to reveal information about other, isolated work-loads running in other containers using different partitions.

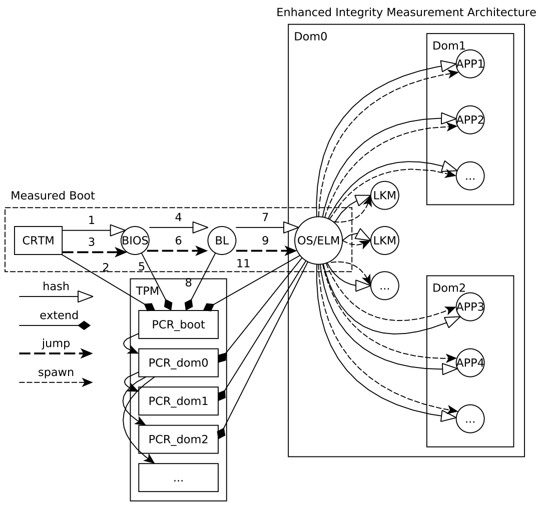

Figure 4. *LS*^{3} Domain Specific Measurements using our Enhanced Loader Module (ELM(m)). The shared system loader program ELM(m) is responsible for spawning new threads. ELM(m) is extended to record programs according to the domain that will be assigned to the thread executing them.

C. EIMA Achieving Security Properties (2) and (5)

In this subsection, we first prove the trustworthiness of the measured boot sequence. We then show how EIMA achieves constrained disclosure which also implies that IMA's integrity measurements are trustworthy. Due to lack of space, we only provide sketches of the proofs.

Definition 1 (Measured Boot Sequence):

$$\begin{aligned}\mathsf{SRTM(m,t)} = {} & \exists t_S, t_B, t_{BL}, t_O, I\ .\ (t_S < t_B < t_{BL} < t_O < t)\\ & \wedge \mathsf{Restart(m,I)}@t_S \wedge \mathsf{Jump(I,BIOS(m))}@t_B\\ & \wedge \mathsf{Jump(I,BL(m))}@t_{BL} \wedge \mathsf{Jump(I,OS(m))}@t_O\\ & \wedge \neg(\mathsf{Restart(m)})\ \mathsf{on}\ (t_S, t] \wedge \neg(\mathsf{Spawn(I)})\ \mathsf{on}\ (t_S, t]\\ & \wedge \neg(\mathsf{Jump(I)})\ \mathsf{on}\ (t_S, t_B) \wedge \neg(\mathsf{Jump(I)\ on}\ (t_B, t_O))\end{aligned}$$

Theorem 1 (Trustworthiness of SRTM Integrity Measurement): Let $seq = \langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m), \text{APP}(m) \rangle$, $\forall t.\ Restart(m, I)@t, \alpha\ :\ m.pcr_{boot} \rightarrow dom_0$, and $\mathsf{Mem(m.pcr.boot, seq)}@t$ then $\mathsf{SRTM(m,t)}@t$.

Proof: Our proofs and programs for CRTM(m), BIOS(m), BL(m), and OS(m) in *LS*^{3} follow [3], except that we use Lemma 1 instead of protected PCRs (see discussion before Lemma 1). ■

So far we have shown that based on the sequence in PCR_{boot} the system must have performed a measured boot sequence to bring up the OS. We now show the trustworthiness of (E)IMA integrity measurements.

Definition 2 (Domain Launch using ELM(m,t)):

$$
\begin{aligned}
&\mathsf{DomainLaunch(m,t)} = \exists t_E, t_{LKM_1}, t_{LKM_2}, t_{App1}, t_{App2}, I, J \ . \\
&\quad (t_E < ((t_{LKM_1} < t_{LKM_2}) || (t_{App1} < t_{App2})) < t) \\
&\quad \mathsf{SRTM(m,t)}@t \wedge \mathsf{Spawn(I,J,ELM(m))}@t_E \\
&\quad \wedge \mathsf{Spawn(J,LKM_1,dom_0)}@t_{LKM1} \\
&\quad \wedge \mathsf{Spawn(J,M_2,dom_0)}@t_{LKM2} \\
&\quad \wedge \mathsf{Spawn(J,App1,dom_1)}@t_{App1} \\
&\quad \wedge \mathsf{Spawn(J,App2,dom_1)}@t_{App2} \\
&\quad \wedge \neg\mathsf{Spawn(dom_0)} \text{ on } (t_E, t_{LKM_1}) \\
&\quad \wedge \neg\mathsf{Spawn(dom_0)} \text{ on } (t_{LKM_1}, t_{LKM_2}) \\
&\quad \wedge \neg\mathsf{Spawn(dom_0)} \text{ on } (t_{LKM_2}, t] \wedge \neg\mathsf{Spawn(dom_1)} \text{ on } (t_E, t_{App1}) \\
&\quad \ldots
\end{aligned}
$$

Theorem 2 (Trustworthiness of EIMA Integrity Measurement): Let $srtm = \langle sinit, \mathrm{BL}(m), \mathrm{OS/ELM}(m)\rangle$, $dom_0 = \langle sinit, srtm, \mathrm{LKM_1}(m), \mathrm{LKM_2}(m), ...\rangle$, $dom_1 = \langle sinit, dom_0, \mathrm{App1}(m), \mathrm{App2}(m), ...\rangle$, $\alpha : m.pcr_{dom_0} \rightarrow dom_0, m.pcr_{dom_1} \rightarrow dom_1$, and at time t we have $\mathsf{Mem(m.pcr.srtm,\ srtm)}$, $\mathsf{Mem(m.pcr.dom_0,\ dom_0)}$, and $\mathsf{Mem(m.pcr.dom_1,\ dom_1)}$, then $\mathsf{DomainLaunch(m,t)}$.

Proof: The proof follows the basic proof of Measured Boot with the same assumptions. Using the sequence in $m.pcr.srtm$ combined with the fact that only the initial thread I executing $\mathrm{CRTM}(m), \mathrm{BIOS}(m), \mathrm{BL}(m), Q \in \mathrm{OS}(m)$ recorded and then spawned $\mathrm{ELM}(m)$ at some time t. The extend operation in $\mathrm{OS/ELM}(m)$ gives us the necessary proof that $\mathrm{OS}(m)$ has in fact reduced `spawn` $\mathrm{ELM}(m)$ at t_E. $\mathrm{ELM}(m)$ continues the measured launch sequence. ■

IV. Conclusion

In this paper, the design and verification of an enhanced integrity measurement architecture (EIMA) was presented. EIMA addresses the trustworthiness and constrained disclosure of integrity measurements for containerized systems. Our development and design was guided by a precise formal model of stratified systems. The formal model was conceived by adding required constructs for domains and domain specific measurements to an existing and established formal model. The additions allowed a proof of a trustworthy measurement architecture from system boot all the way up to containerized applications. The proof of EIMA has also shown the security properties of the commonly used IMA. However, EIMA does not reveal information about a target container to other untrusted tenants on the same system. Future work will have to address more detailed interactions between otherwise isolated domains and sub-domains via system calls and privileged administrator commands. The verification of properties which were deferred to the TPM is a pressing task and a suitable remote attestation protocol is still needed.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/945445.945462

[2] T. Braüner, *Hybrid Logic and its Proof-Theory*. Dordecht, Germany: Springer-Verlag, 2011.

[3] A. Datta, J. Franklin, D. Garg, and D. Kaynar, "A logic of secure systems and its application to trusted computing," in *2009 30th IEEE Symposium on Security and Privacy*, May 2009, pp. 221–236.

[4] D. Devriese, L. Birkedal, and F. Piessens, "Reasoning about object capabilities with logical relations and effect parametricity," in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, March 2016, pp. 147–162.

[5] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson, "The digital distributed system security architecture," in *National Computer Security Conf., NIST/NCSC, Baltimore*, vol. 12. National Institute of Standards and Technology, January 1989, pp. 305–319.

[6] ISO, "Trusted platform module library," International Organization for Standardization, Geneva, Switzerland, ISO ISO/IEC 11889-1:2015, 2015.

[7] L. Jia, S. Sen, D. Garg, and A. Datta, "A logic of programs with interface-confined code," in *2015 IEEE 28th Computer Security Foundations Symposium*, July 2015, pp. 512–525.

[8] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," *arXiv e-prints*, p. arXiv:1902.03383, Feb 2019.

[9] Y. Juglaret, C. Hritcu, A. A. D. Amorim, B. Eng, and B. C. Pierce, "Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, June 2016, pp. 45–60.

[10] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 361–372.

[11] H. Lauer and N. Kuntze, "Hypervisor-based attestation of virtual environments," in *Advanced and Trusted Computing (ATC), 2016 Intl IEEE Conferences*. IEEE, 2016, pp. 333–340.

[12] H. Lauer, C. Rudolph, A. Salehi S., and S. Nepal, "User-centered attestation for layered and decentralized systems," in *Network and Distributed Systems Security (NDSS) Symposium 2018, Workshop on Decentralized IoT Security and Standards (DISS), 18-21 February 2018, San Diego, CA, USA*. ISOC, 2018.

[13] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.

[14] A. Rein, "Drive: Dynamic runtime integrity verification and evaluation," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: ACM, 2017, pp. 728–742. [Online]. Available: http://doi.acm.org/10.1145/3052973.3052975

[15] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a tcg-based integrity measurement architecture." in *USENIX Security Symposium*, vol. 13, 2004, pp. 223–238.

[16] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-sealed data: A new abstraction for building trusted cloud services," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 175–188. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/santos

[17] A. Sinha, L. Jia, P. England, and J. R. Lorch, "Continuous tamper-proof logging using tpm 2.0," in *Proceedings of the 7th International Conference on Trust and Trustworthy Computing - Volume 8564*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 19–36. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08593-7_2

[18] J. Son, S. Koo, J. Choi, S.-j. Choi, S. Baek, G. Jeon, J.-H. Park, and H. Kim, "Quantitative analysis of measurement overhead for integrity verification," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17. New York, NY, USA: ACM, 2017, pp. 1528–1533. [Online]. Available: http://doi.acm.org/10.1145/3019612.3019738

[19] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2435–2450. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134098