Abstract. This short document collects the formalism behind the attestation design maxims and explains the classes of attacks the maxims are intended to prevent. The human-readable maxim statements are kept in ‘maxims.tex‘; the technical formalism and examples appear below.

The Attestation Maxims Formalism

No Author Given

No Institute Given

1 Overview

The following pages present the formal model we use for attestation analysis and explain how each maxim maps to classes of attacks. Macros providing the human-readable maxim text are defined by ‘maxims.tex‘, which is intentionally loaded before the document body so that those macros may be referenced from the included fragments.

2 Attacks Prevented by Following the Maxims

Why should one follow the attestation design maxims prescribed by this paper? Each maxim is specifically designed to prevent certain classes of attacks on attestation systems. Importantly, we do not claim that *all* attacks are prevented by following these maxims; rather, each maxim prevents attacks that would be viable if the maxim were not followed.

We must start by abstractly defining the system that we are protecting:

Definition 1 (Measurable System). *A* measurable system *is a system made of the following:*

- *A set of* components C
- *A set of measurers* $M \subseteq C$
- *A state space* S*: each component* $c \in C$ *has a state* $s_c \in S$ *The possible states that a component can be are:*
  - Safe*: the component is operating correctly and as intended.*
  - Corrupt L*: the component is operating incorrectly or maliciously by some corruption with label* L*.*
  - Infected L*: the component is operating correctly, and has been compromised by a past corruption with label* L*.*
- *A dependency relation* $D \subseteq C \times C$ *where* $(c_1, c_2) \in D$ *means that component* c_1 *depends on component* c_2*. Intuitively, this means that if* c_2 *is corrupt, then* c_1 *may also be corrupt.*

We them define the runtime of the system as follows: There is a sequence of events $E = [e_1, e_2, \ldots, e_n]$ that occur in the system.

Each event e_i is one of the following:

- A *boot event* `Boot` where all components are initialized. *Notably*, the boot event will always be e_1.

- A *protocol start event* `Start` where the attestation protocol begins.
- A *protocol end event* `End` where the attestation protocol ends.
- A *launch event* `Launch`(c) where component $c \in C$ is launched.
- A *measurement event* `Meas`(c) where measurer $m \in M$ measures component $c \in C$.
- A *corruption event* `Corrupt`(c, L) where component $c \in C$ is corrupted with label L.
- A *repair event* `Repair`(c) where component $c \in C$ is repaired to the safe state.
- An *infection event* `Infect`(c_1, c_2, L) where component $c_2 \in C$ is infected with by component c_1 with label L.
- A *use event* where all of the components are used for their intended purpose and the security critical operations are performed. *Notably*, the use event will always be the last event: e_n.

We will consider *corruption, repair, and infection events* to be adversarial events that are performed by an attacker trying to compromise the system. *Protocol start,end, and measurement events* are performed by the attestation system to measure the state of components. *Boot, launch, and use events* are benign system events that occur naturally in the system.

We denote event widths in the standards mathematical notation for an interval.

Definition 2 (Timely Attack). *A* timely attack *is an attack where the attacker performs a corruption event* `Corrupt`(c, L) *on component c after the measurement event* `Meas`(c) *where component c is measured, but before the* `Use` *event.*

Thus, any timely attack must fall within the interval $($`Meas`$(c),$ `Use`$]$. *Intuitively, this is because a component that is corrupted* before *the measurement event will be detected by the measurement event, and a component that is corrupted* after *the use event does not affect the security of the system during its use.*

We consider *timely attacks* to be acceptable attacks that cannot be prevented by any attestation system, as they occur after the measurement has been performed.

As a side note however, timely attacks can be arbitrarily difficult to perform in practice, given the interval of attestations that occur and the time between attestation and use. Overall, this is the well-known *time-of-check to time-of-use* (TOCTOU) problem; to which we leave solutions for other work.

Ultimately, we then define the goal of our attestation system as follows:

Definition 3 (Attestation Goal). *The goal of the attestation system is to detect all attacks that are not timely attacks (i.e., all attacks that occur before the measurement event) or deep attacks (i.e. attacks that compromise a root of trust).*

We define detection of an attack as follows:

Definition 4 (Attack Detection). *An attack on component c with label L is detected if there exists a measurement event $\texttt{Meas}(c)$ that occurs after the most recent corruption event $\texttt{Corrupt}(c, L)$ on component c with label L, and before the use event $\texttt{Use}$.*

Essentially, the following trace must exist:

$$\ldots, \texttt{Corrupt}(c, L) + +others + +\texttt{Meas}(c), \ldots, \texttt{End}, \ldots \texttt{Use}$$

Where $Rep(c) \notin others$ (i.e., there is no repair event on component c between the corruption and measurement events).

TODO: Add the events semantics here

We can now define how the maxims prevent attacks on the system.

??. *Constrain possible interactions so attested properties depend only on limited, predictable, measurable parts of the system.* //

The maxim of constraining interaction prevents attacks where a measurer's output depends on unmeasured or unpredictable components.

For example: **If** some c_i exists such that the overall system security goal G depends on the state of c_i, but c_i is not measured by any measurer $m \in M$ during the attestation protocol, **then** an attacker can perform a corruption event $\texttt{Corrupt}(c_i, L)$ on component c_i before the measurement event $\texttt{Meas}(c_i)$ (as $\texttt{Meas}(c_i)$ does not exist in this scenario) thus violating the attestation goal, as the attack is not detected. **TODO: Define G formally above**

Concretely, the following trace would exist:

$$\ldots, \texttt{Corrupt}(c_i, L) + +others + +\texttt{End}, \ldots \texttt{Use}$$

Where $Meas(c_i) \notin others$ (i.e., there is no measurement event on component c_i).

Ultimately, this expands the attack window from $(\texttt{Meas}(c_i), \texttt{Use}]$ to $(\texttt{Launch}(c_i), \texttt{Use}]$, allowing the attacker to perform the corruption at any time after the component is launched. **TODO: Go into how Launch must proceed Meas**

??. *Short-lived, single-input processes avoid the need for process runtime remeasurement that long-lived services require if they handle untrusted inputs.* //

??. *Attestation should be independent of secrets that could be disclosed by transient corruptions, lest transient corruptions cause permanent attestation failure.* //

??. *Each attestation must display characteristics showing that it originated from our uncorrupted attestation software, running under a trustworthy system boot.* //

??. *An acyclic dependency relation enables us to measure lower layers before components depending on them. Only very recent corruptions of underlying components can then undermine attestations, meaning corruptions during the course of attestation.* //

3 Concluding Remarks

The maxims are deliberately concise design principles. The formalism above provides a minimal model for reasoning about when an attestation is likely to detect an attack and when an attack falls into the time-of-check-to-time-of-use window. Future work may expand the event semantics and provide automated proofs using the Coq development contained in the ‘theories/‘ directory.