Provenance-Based Model for Verifying Trust-Properties

Cornelius Namiluko and Andrew Martin

Oxford University Department of Computer Science,
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
firstname.lastname@cs.ox.ac.uk

Abstract. Trust establishment requires evidence about the system's ability to operate as expected. However, the nature of this evidence and its representation and usage in trust evaluation still remains an open problem. Current mechanisms for collecting this evidence, such as the TCG integrity schema, do not support the linkage of this evidence and therefore limit the kinds of properties that can be verified. We argue that provenance provides more comprehensive evidence that can be represented in a manner that eases trust evaluation. Towards this end, we propose a *provenance-based* model for reasoning about a system's ability to satisfy trust properties of interest. This approach enables interoperability, supports multiple abstractions and enables evaluation of varying trust properties. Its application on verifying properties of platforms for use in a trust domain demonstrate its feasibility and flexibility.

1 Introduction

Distributed systems have the potential to deliver cheaper, flexible and scalable computation and data storage solutions. However, security and trust still pose a significant challenge towards their wider adoption [3]. This subject has received considerable attention and several systems that use trusted computing [1,10] have been proposed to address this challenge. These systems provide information about their configurations, which can be used to determine whether or not the system's behaviour conforms to expectations. However, the question of what information is necessary, how it can be represented and how it can be used in trust evaluation is still an open problem.

We argue that provenance provides more comprehensive evidence (including integrity of the components and activities that occur on the system such as events, processes, interactions e.t.c.) which can be captured in a manner that eases trust evaluation. Towards this end, and motivated by the realisation that trusted computing and provenance seek to address similar issues [7], we propose a *provenance-based model* which captures activities on a system as a *provenance graph*. The model extends the Open Provenance Model (OPM) [9] to enable provenance to be captured in a manner that supports verification of trust properties.

S. Katzenbeisser et al. (Eds.): TRUST 2012, LNCS 7344, pp. 255–272, 2012.

This approach has a number of advantages including: i) ability to provide a more comprehensive view of a system; ii) support for multiple abstractions; iii) support for interoperability; and iv) support for varying complexity in the types of properties that can be expressed. We apply this approach to the verification of platforms for use in a *trust domain*.

1.1 Virtual Platforms for Trust Domains

Information sharing is crucial for any successful collaboration. However, the sensitive nature of certain information may prevent or discourage entities from sharing it. To overcome this challenge, the Trust Domains Project[1] proposes the concept of a *trust domain* as a means of capturing the state and processes that allow information to be shared among entities that exhibit shared and predictable behaviour to protect the information.

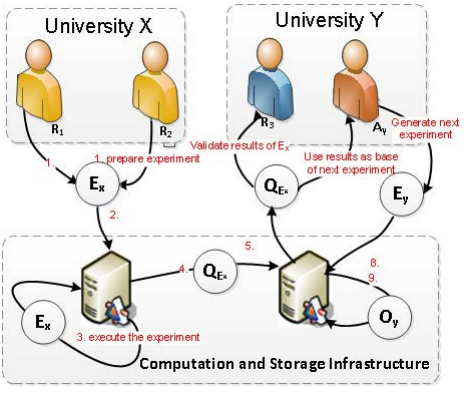


Fig. 1. A shared infrastructure used to execute experiments. E_x is an experiment set-up and Q_{E_x} is the completed experiment containing results.

Figure 1 illustrates an application of such a concept. In this scenario, researchers collaborate on a number of projects, each of which is comprised of a number of experiments. Researchers from university X may create an experiment which might be validated or used in the next series of experiments by other researchers from Y. To facilitate this, experiments are created as virtual appliances (VA) — ready-to-use virtual machine images configured with an operating system and a software stack necessary for a particular experiment. However, since the virtual appliances are created outside of their control, researchers need to establish whether or not virtual appliances will enforce appropriate data flow control before entrusting them with data for the experiment. This can be achieved by collecting evidence that could support a VA's behavioural characteristics and representing such evidence in a manner suitable for trust evaluation.

[1] The Trust Domains project is a TSB and EPSRC funded project that aims to build a framework for controlled information sharing. Further details are available on `http://www.hpl.hp.com/research/cloud_security/TrustDomains.pdf`

2 Related Work

Our work is motivated by Lyle and Martin [7] who note that provenance and trusted computing could complement each other. We build on our previous work [4] on a trace-based model for verifying properties of virtual appliances and on an open provenance model (OPM) [9]. This work proposes the semantics of a model that can be used to capture trust-relevant evidence. It differs from our previous work in that it is more general, i.e. the trace of events considered is in fact a subset of provenance, and has a simplified means of specifying trust properties, as opposed to the CSP specifications mechanism proposed in [4].

The idea of collecting provenance from virtual appliances has also been investigated by Wei [8]. Our work differs in that we are interested in the evaluation of trustworthiness of a system using the generated provenance rather than the trustworthiness of provenance records. The idea of developing a more comprehensive view of the system configurations has also been discussed in [14,2]. Presti [14] proposes the notion of a tree of trust as a mechanism for representing verification data. Schmidt et. al. [2] builds on this structure and proposes modifications to the TPM command set and data structures to support the derivation of tree-formed verification data. In our case, the TPM does not require any modifications. Instead it is simply used to validate the authenticity of the nodes in the graph. Whereas in a tree structure there is only one way of getting to a particular node, our graph-based approach provides a richer semantics — enabling verifiers to consider multiple paths to a node. Furthermore, our model is extensible and supports interoperability.

The TCG Infrastructure Working Group recognises that trust establishment must consider the origin, condition and history of components used to construct the platform. However, the proposed architecture[2] is limited to capturing components that exist on a system rather than how those components interact or how they are related. Our approach is more comprehensive in that it includes the activities that occur on the system in question and relations among components involved in those activities.

3 Trust Properties and Evidence

The TCG defines trust in terms of the expectations of a relying party on the behaviour of the system they wish to rely on. These expectations can be considered constraints on the behaviour of the system being relied on. We call these constraints *trust properties* and define them as *constraints that capture a trustor's expectations on the behaviour of the system.* But what kinds of constraints are necessary to arrive at a particular trustworthiness decision? What kind of information is necessary to support such decisions?

The answers to these questions will depend on a number of factors such as the level of trust desired, the amount of information available and the ability

[2] `http://www.trustedcomputinggroup.org/files/resource_files/87651761-1D09-3519-AD6C5B3E41547285/IWG_ArchitecturePartII_v1.0.pdf`

of the trustor to use this information. In this section, we describe the kind of information, which we refer to as *evidence* that can be collected in the scenario described in Section 1.1 and the trust properties applicable to it.

3.1 Trust Properties

We identify four categories of trust properties as follows:

1. *Possible future behaviour*: seek to determine whether or not a VA will exhibit certain behavioural patterns when executed. Examples include: i) an executable will use known configurations; ii) a given executable will run before another executable; or iii) cryptographic keys will be reset at start-up.
2. *Processes performed and parameters used*: identify the processes carried out during the creation of a VA, the order in which they were performed and the parameters that were used as input to the processes. Examples include: i) a certain package was installed; ii) a package was configured as expected; or iii) certain privileges were assigned to a given object.
3. *Data sources and integrity*: seek to establish the authenticity of data such as packages included on a VA. Some examples include: i) packages installed were downloaded from trusted sources; ii) all critical packages installed were of a known integrity; or iii) a given file was obtained from a known package.
4. *Integrity of processes*: seek to determine whether or not the software components that executed as part of the build process, described below, behaved as expected. These may include: i) the executed programs have known integrity values; or ii) a particular process used the expected executable files.

3.2 Evidence Classification

The properties discussed above can be determined by collecting evidence from three main sources: *build platform*; *build process*; and *verification meta-data*.

1. *Build platform* — provides services for creating VAs and thus determines the behaviour of the resulting VA. Evidence from the build platform may include: i) components executed e.g. VA build tools, package managers; ii) configurations of the components e.g. ports open, digital signature checks or enabled services; or iii) dependency resolution among components e.g. versions of libraries used by the components.
2. *Build process* — involves a number of steps including package download and installation, configuration changes and execution of specified scripts. Evidence from this process might include: i) integrity values of the input and output e.g. command line parameters, environment variables; ii) configuration settings for virtual appliance, e.g. user accounts and privileges, network configurations and start-up scripts; and iii) virtual appliance contents e.g. software packages installed or files copied to the disk image.
3. *Verification metadata* — provides information about the format or validity of other pieces of evidence. Examples include: i) integrity schemas and reference manifests; ii) digital signatures; and iii) meta-data about the repositories from where packages are downloaded.

4 Graph-Based Representation

To enable meaningful trust evaluation, the evidence, discussed above, must be captured in an interoperable manner (since the producer may be different from the consumer of the evidence) and must include relationships among the parts of the evidence. Towards this end, we propose a provenance-based model that extends the open provenance model (OPM) [9] to capture the evidence.

The model is based on the idea that information about the data used, processes performed, entities that perform these processes and any new data generated is captured as a set of RDF triples, where each triple (X, Y, Z) specifies that a component X was related to another component Z through the property Y. In the rest of this section, we describe the extensions to OPM necessary to support reasoning about trust properties.

4.1 A Summary of OPM Semantics

Since our model builds on OPM, we begin with a summary of the main semantics of OPM, a detailed discussion of which can be found in Groth and Moreua [9].

OPM defines three main entities in a provenance record. These include: *Agent*, *Artifact* and *Process*, where an agent is an entity capable of performing a process, an artifact is an immutable piece of state and a process is a series of actions that use artifacts and generate new artifacts. These entities are related through a number of properties as depicted in Figure 2. The *wasTriggeredBy* (WTB) defines a relationship in which one process is made operational by another process. A process can be specified to have been controlled by multiple agents through the *wasControlledBy* (WCB) property. Artifacts used in a process are indicated through the *used* property while those that are created by a process are related to the process that created them through the *wasGeneratedBy* (WGB) property. The *used* and *WGB* properties must occur after the process has been created. To maintain the link between those artifacts that are used and those created, the *wasDerivedFrom* (WDF) property is used to specify that one artifact was derived from another. However, these semantics introduce some limitations (as discussed in the following sections) towards capturing the evidence for the purpose of trust evaluation.

4.2 Program Execution

A program can be captured as an agent in OPM. However, the semantics of the properties defined in OPM limit the ability to express execution relationships

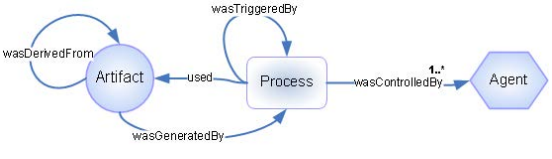


Fig. 2. An illustration of the main components of OPM. Artifacts are illustrated with a circle, agents by a hexagon and processes by rectangles.

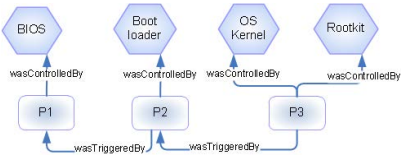
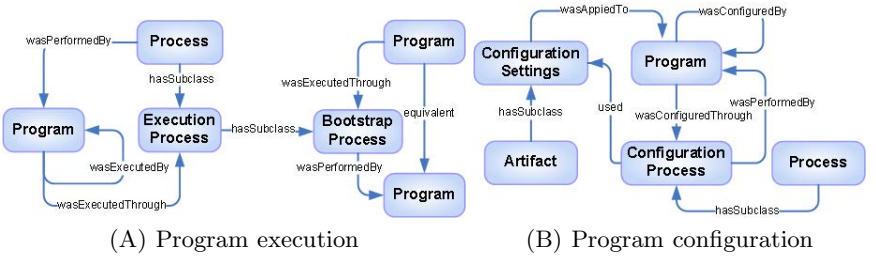


Fig. 3. Capturing the boot phase using OPM. Shows that a process can be controlled by multiple programs.

among programs. Consider the boot phase of a system in which the BIOS executes the boot loader, which in turn executes the operating system kernel. Using OPM, this scenario can be captured as illustrated in Figure 3. The BIOS controls a process, $P1$, which triggers another process, $P2$, controlled by the boot loader. $P2$ triggers $P3$, which is controlled by the operating system kernel. The semantics of the *WCB* property, however, imply that a single process can in fact be controlled by multiple programs (e.g. $P3$ *wasControlledBy* OS Kernel and Rootkit in Figure 3). Alternatively, the concept of *role* defined in OPM could be used to specify the role played by each program linked through the *WCB* property. However, roles are defined as labels and would still require a similar effort in defining semantics to make them useful in trust evaluation.

An alternative approach would be to use the TCG integrity architecture to create a chain of trust which captures the notion that a program in the chain executed and transferred control to the next program in the chain. However, such an approach is not sufficient to capture the idea that other activities could have been happening at the same time as the execution of programs in the chain. To overcome these limitations, we propose an extension to OPM that captures aspects about program execution. The extension, illustrated in Figure 5.A, enables programs to be related to the processes through which they are executed and the component executing them. This has an advantage that when establishing trust not only is the resulting chain of execution checkable, but also the processes in which the chain was created. We formally define the extension to include *ExecutionProcess* and *BootstrapProcess* processes and a number of properties that relate programs to the processes through which they were executed



(A) Program execution (B) Program configuration

Fig. 4. Conceptual representation of extensions to support program execution and configuration

as well as to the program that initiated the execution (note: *ran* is a *Z* function which returns the range of a given relation).

$$
\begin{array}{l}
wasPerformedBy : Process \rightarrow Program \\
ExecutionProcess : \mathbb{P}\, Process \\
wasExecutedThrough : Program \rightarrow Process \\
wasExecutedBy : Program \rightarrow Program \\
wasExecutedAt : Program \rightarrow Time \\
BootstrapProcess : \mathbb{P}\, ExecutionProcess \\
\hline
ExecutionProcess = \operatorname{ran} wasExecutedThrough \\
\forall\, p1, p2 : Program \bullet \\
\quad wasExecutedBy\ p2 = p1 \Leftrightarrow (\exists\, e : ExecutionProcess \bullet \\
\qquad wasPerformedBy\ e = p1 \wedge wasExecutedThrough\ p2 = e) \\
\forall\, e : ExecutionProcess \bullet \\
\quad e \in BootstrapProcess \Leftrightarrow (\exists\, p1 : Program \bullet \\
\qquad wasPerformedBy\ e = p1 \wedge wasExecutedThrough\ p1 = e)
\end{array}
$$

Intuitively, an *ExecutionProcess* is performed by a specific program and yields another program (i.e. the new program goes into the running state). A special type of execution process in which the program that performs the execution is the same as the program that is yielded is referred to as a *BootstrapProcess*.

4.3 Program Configuration

Configuration settings play an important role in determining the behaviour of a program. Grawrock [5] notes that in any non-trivial system, there will be a number of configuration options that may affect how a system behaves. For this reason, configuration settings used in a system must be considered when evaluating the system's trustworthiness.

In OPM, configuration settings can be captured as a type of *Artifact*. This artifact can then be linked to the process that uses it through the *used* property to capture the idea that a process is configured with the configuration settings specified. However, as discussed in the previous section, the semantics of *WCB* imply that configuration settings used by a process cannot be linked to the specific program being configured because multiple programs are linked to the same process that uses the artifact. Furthermore, the semantics of *used* imply that a process has to start its operation *before* it can be configured. However, for trust evaluation it is important to capture the idea that a program was configured in a certain way before it engaged in some other activities. To achieve this, we define an extension to OPM, illustrated in Figure 5.B, in which *ConfigurationSettings* is defined as a type of artifact that can be used to configure a program in a process called *ConfigurationProcess*. We capture this extension formally as follows.

$$
\begin{array}{l}
ConfigurationSettings : \mathbb{P}\, Artifact \\
ConfigurationProcess : \mathbb{P}\, Process \\
wasAppliedTo : ConfigurationSettings \mapsto Program \\
wasConfiguredBy : Program \mapsto Program \\
wasConfiguredThrough : Program \mapsto ConfigurationProcess \\
\hline
\forall\, p : Process \bullet p \in ConfigurationProcess \Leftrightarrow \\
\quad (\exists\, c : ConfigurationSettings \bullet c \in used(\!|\, p \,|\!)) \\
\forall\, c : ConfigurationSettings;\ p : Program \bullet p \in wasAppliedTo\, (\!|\, c \,|\!) \Leftrightarrow \\
\quad (\exists\, e : ConfigurationProcess \bullet c \in used(\!|\, e \,|\!) \land e \in wasConfiguredThrough(\!|\, p \,|\!)) \\
\forall\, p1, p2 : Program \bullet p1 \in wasConfiguredBy(\!|\, p2 \,|\!)) \Leftrightarrow \\
\quad (\exists\, e : ConfigurationProcess, c : ConfigurationSettings \bullet \\
\qquad wasPerformedBy\ e = p2 \land p1 \in wasAppliedTo\, (\!|\, c \,|\!))
\end{array}
$$

This extension allows us to capture properties such as "a program X configured another program Y with settings Z".

4.4 Integrity Measurement

Integrity measurement can be captured as a process using OPM, so that the entity being measured is linked to the integrity measurement process through the *used* property while the resulting integrity value is linked to the process that performs the measurement through the *WGB* property. However, the semantics of *used* imply that only artifacts can be integrity measured because the used relationship can only be applied to artifacts. To overcome this limitation, we introduce an extension to OPM, illustrated in Figure 5.A and formally defined below, which includes a type of process referred to as *IntegrityMeasurementProcess*, an artifact called *IntegrityValue*, a *Measurable* type and a number of properties that relate these concepts (and those already defined in OPM).

$$
\begin{array}{l}
IntegrityMeasurementProcess : \mathbb{P}\, Process \\
performedOn : IntegrityMeasurementProcess \rightarrow Measurable \\
IntegrityValue : \mathbb{P}\, Artifact \\
integrityOf : IntegrityValue \nrightarrow Measurable \\
measured : Program \mapsto Measurable \\
\hline
Measurable = (Artifact \cup Agent) \\
\forall\, p : Program;\ m : Measurable \bullet measured\, p = m \Leftrightarrow \\
\quad (\exists\, e : IntegrityMeasurementProcess \bullet performedOn\ e = m \\
\qquad \land\ performedBy\ e = p)
\end{array}
$$

The extension specifies that *IntegrityMeasurementProcess* is a process that can take agents, in addition to artifacts, as input and produce another artifact of type *IntegrityValue*. The entity whose integrity is being taken can be linked to the resulting integrity value through the *integrityOf* property and to the *IntegrityMeasurementProcess* through the *performedon* property.

4.5 Communication

Components on a system communicate through various means to provide services to one another (e.g. remote procedure calls), inform each other about their activities or observations (e.g. events, message broadcasts) and exchange information for use in computations. For example, in the scenario described in Section 1.1 a package manager communicates with the repository to download packages for installation on a virtual appliance. This communication creates interactions which determine the flow of information within and across systems. Of particular interest to the scenario is the ability to track the source of packages that are installed on a VA.

To cater for communication aspects of the system, we make use of a combination of the D-profile proposed by Groth and Moreau [13] and the common module defined for the open provenance model vocabulary (OPMV)[3]. The D-profile defines the relationship between a sender process and the message it sends as well as the receiver process and the message it receives. This is useful for capturing communication. However, if we need to capture interactions resulting from this communication, the D-Profile falls short. To solve this, we complement it with concepts defined in the common module. More specifically, we use *Download*, *downloadUri*, *connection* defined in the namespace, *http://purl.org/net/opmv/types/common#* and *Connection* defined in the namespace, *http://www.w3.org/2006/http#*.

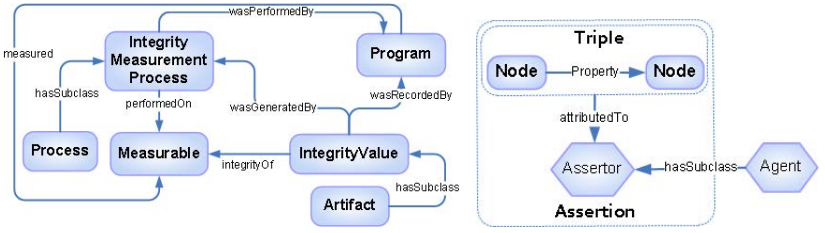
4.6 Assertion Model

We have so far defined a way of capturing the various kinds of evidence as provenance statements. However, these statements are only sound if they can be traced to a *root of trust*. Groth and Moreau [13] note that in distributed systems, where there may be multiple monitoring and reporting components, each provenance entry must be linked to the entity that reports it. They introduce the *attributedTo* property to link an account of provenance to an entity responsible for it. However, the soundness of these attributions can only be determined if a link can be created between the attributing (*assertor*) entity and a root of trust, i.e. a root of trust for assertion.

To achieve this, we propose an extension to OPM, called *Assertion Model*, which makes use of the earlier defined extensions to create links between the assertor to other components which could potentially serve as roots of trust. In other words, this can be used to create multiple chain of trust (based on different aspects of the system, not just the measure-before-load [5] as is the case for TCG based chain of trust) with the assertor at one end of the chain and a root of trust for assertion at the other end.

In this model, each triple (s, p, o) is linked to an assertor to create a pair $(A, (s, p, o))$, which we refer to as an *assertion*. This specifies that an agent A

[3] Common Module is a specialisation of OPMV that defines commonly used terms not defined in the OPM specifications, see `http://code.google.com/p/opmv/wiki/GuideOfCommonModule` for details.



(A) Integrity measurement (B) Structure of Assertion

Fig. 5. (A) depicts the integrity measurement extension while (B) shows the structure of an assertion for use in the assertion model

asserts that a subject s is related to an object o through a property p. Figure 5.B depicts how each assertion is captured.

$$Property == \{wasConfiguredBy, wasPerformedBy, wasExecutedThrough, wasExecutedBy, ...\}$$

$$\begin{array}{|l} Assertor : \mathbb{P}\, Agent \\ Assertion : (Assertor \times (Node \times Property \times Node)) \\ attributedTo : (Node \times Property \times Node) \rightarrow Assertor \\ occuredAt : (Node \times Property \times Node) \rightarrow Time \\ \hline Node = (Process \cup Agent \cup Artifact) \\ \forall\, x, y : Node;\ p : Property;\ a : Assertor \bullet \\ \quad (a, (x, p, y)) \in Assertion \Leftrightarrow y \in p(\!|\, x \,|\!) \wedge attributedTo\,(x, p, y) = a \end{array}$$

5 Reasoning about Trust-Properties

The graph representation discussed above provides a means of representing the evidence about the activities on a system. But how useful is this evidence and how does the graph representation help in trust evaluation? This section addresses these questions by proposing an approach in which the evidence is validated against a set of criteria which aim to determine its soundness before verifying it to determine if certain properties can be satisfied by the graph.

5.1 Evidence Validation

The evidence presented in a provenance graph can come from multiple sources. As discussed in Section 4.6, for the purpose of trust evaluation, this evidence must be linked to the entities that generate it. Therefore, we consider evidence to be valid if it can be liked to an entity that can be securely identified. The use of the assertion model simplifies this by providing a link between the assertions and the assertors so that validation is based on the ability to securely identify assertors for either the entire set of evidence or a subset of it. To achieve this, we

develop three validation rules, which when taken together specify that evidence presented in a particular subset of the graph is sound.

$$G = \{(A, (s, p, o)) \mid s, o \in Node;\ p \in Property;\ A \in Assertor\}$$

1. *RULE 1*: each assertion (s, p, o) must have been asserted by some agent that exists within the system (SystemComponents is a set of components in a system's architecture).

$$\forall\, a : G \mid valid(a) \Leftrightarrow a.1 \in SystemComponents$$

2. *RULE 2*: if the assertor is a program, then it must have been executed before performing the assertion.

$$\forall\, g : G \mid g.1 \in Program \bullet \\ valid(a) \Leftrightarrow wasExecutedAt\ a.1 < occuredAt\ a.2$$

3. *RULE 3*: the assertor must be securely identifiable (this does not necessary mean that the identity is the expected one, this is checked during verification).

$$\forall\, g : G \mid valid(g) \Rightarrow (\exists\, iv : IntegrtiyValue \bullet integrityOf\ iv = g.1)$$

5.2 Property Specification and Verification

Our verification model is based on RDF graph pattern matching provided in SPARQL [6]. First, each property specification, discussed in the previous section, is captured as a basic graph pattern (BGP) — a set of triples which may have some of the elements represented by variables. Then the obtained BGP is mapped to the graph (or a sub-graph) to determine if there is an entailment relationship between the BGP and the graph. In the remainder of this section, we discuss how properties are specified and verified on a given graph.

Presence/Absence of Triple Patterns: properties can be specified in terms of the presence or absence of certain triples. For example, to specify that a firewall was installed, one can check the graph to determine whether or not the triples $(installProcess, wasPerformedBy, rpm)$ and $(installProcess, used, firewall.rpm)$ exist. This is achieved by specifying the triples to be checked as a BGP (when the values of the triple elements are important) or graph templates (when certain values can be ignored). Verification is achieved by performing a query in the form of *ASK*, which returns true or false, depending on whether the specified triples can be found in the graph. Listing 1.1 shows how the example of firewall installation can be verified.

Listing 1.1. Example query for determining presence/absence of triples

```
ASK
    { :installProcess :wasPerformedBy :rpm . }
    { :installProcess :used :firewall.rpm . }
```

Values of Triple Elements: elements of a triple have values which can be used to infer certain information about the behaviour of a system. In the TCG-based integrity mechanism, for instance, trust is based on the presence of components with known integrity values. We support specification of triple element values using the FILTER feature of SPARQL. For example, to determine that in a given execution, the installed firewall had a certain integrity value, a query such as that shown in Listing 1.2 can be specified.

Listing 1.2. Example query for determining triple element values

```
ASK
    { :installProcess    :used          :firewall.rpm .
      ?iv                :integrityOf   :firewall.rpm .
    FILTER ( ?iv = "cdf84324"^^xsd:string )}
```

Supporting Multiple Abstractions: the property that any set of triples can be defined as a graph enables us to provide multiple abstractions. So that a subgraph that concentrates on certain types of assertions can be obtained from a provenance graph. This is achieved by specifying a graph template that includes properties useful for a certain abstraction. For example, to capture a graph that can be used to determine an execution chain of trust on a system, a graph template such as *?x :wasExecutedBy ?y* can be used to return all triples that have a *wasExecutedBy* property between any two entities (represented by the variables x and y). Such a graph can be obtained by using the *CONSTRUCT* query form on a provenance graph, which returns a graph matching the triples specified. For example, the query in Listing 1.3 returns a graph which only includes assertions related to programs executed on a system. The resulting graph can be subjected to further analysis as discussed above.

Listing 1.3. Example query to create abstraction for executions

```
CONSTRUCT    { ?x ?p      ?y .}
WHERE        { ?x :wasExecutedBy ?y .}
```

Sequencing of Triples: in most cases, a triple taken in isolation does not provide much information. To develop a more meaningful judgement of the behaviour of the system requires a way of relating the triples. One such relationship is the sequencing of triples. For example, to specify that a given program was configured in a certain way before it participated in a process, would involve checking that the program configuration occurred before a particular process was performed. Triple sequencing can be specified using the assertion model, where each assertion is linked to the time instance at which a particular triple occurred, using the *occuredAt* property. Given a sequence of triples $T =< t1, t2, ...tn >$, the FILTER construct can be used to relate the times at which each triple occurs. Listing 1.4 shows an example (s = subject, p = property and o = object).

Listing 1.4. Example query to determine sequence of triples

```
ASK
    {t1.s        t1.p              t1.o .
    t1.(s,p,o)   :occuredAt        ?x1 .
    t2.s         t2.p              t2.o .
    t2.(s,p,o)   :occuredAt        ?x2 .
    ...
    tn.s         tn.p              tn.o .
    tn.(s,p,o)   :occuredAt        ?xn .
    FILTER ( ?x1 < ?xn <...<?xn )}
```

6 Verifying Virtual Appliances

In this section, we describe an experiment that demonstrates how our proposed model can be used to verify trust properties of platforms before they can be admitted into a trust domain.

6.1 Set-Up and Provenance Collection

The experiment was set-up as illustrated in Figure 6. A build platform was setup using openSUSE 11.3 running on a kernel compiled with IMA support. Kiwi imaging system[4] and *strace*[5] packages were installed and a simple shell script was set-up to execute Kiwi using *strace* as a tracing tool. Execution traces were collected including log files from Kiwi, integrity measurement log (an extract of which appears in Listing 1.5) and a trace generated by *strace* (an extract appears in Listing 1.6), which were processed and verified on a separate platform.

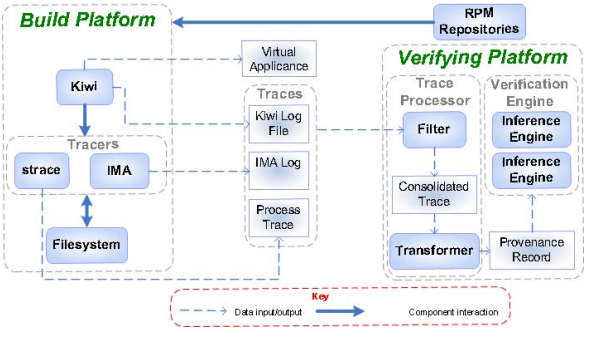


Fig. 6. An experimental setup for generating provenance for virtual appliances that will be used in a trust domain

[4] Kiwi is a tool, from openSUSE Build Services, used for creating VM images.
[5] Strace is a UNIX tool for tracing system calls.

Listing 1.5. Example IMA log showing a selected list of log entries with hash values truncated to six digits

```
10 000000 ima 000000 boot_aggregate
10 7ba06b ima 095baf /init
...
10 ba0922 ima bb4476 ./build.sh
10 f80069 ima e54a84 /usr/bin/strace
10 f97bd8 ima e10ec0 /usr/sbin/kiwi
....
10 1125be ima 908990 config.xml
10 b97636 ima 185cb1 KIWIConfig.sh
....
10 6b81fa ima cc622c /usr/bin/zypper
....
10 988080 ima bddd59 zypper.conf
10 229eb6 ima 6bf998 opensuse.org_distr_xx_.repo
...
10 2d6856 ima 3b2310 openSUSE-xx-.i586.rpm
10 e828ee ima b0d515 filesystem-xx-.i586.rpm
10 827556 ima bc4d2a vim-base-7.xx.i586.rpm
....
10 d9545c ima f56808 config.sh
```

Listing 1.6. Example entries in strace output

```
3056 18:19:08 clone(child_stack=0,....) = 3057
3057 18:19:08 execve("/usr/bin/zypper", ....) = 0
...
3057 18:19:42 clone(child_stack=0,....) = 3069
3069 18:19:42 execve("/bin/rpm",..,http:download.
    opensuse.org..openSUSE-xx-.i586.rpm
...
4354  18:21:51 open("/tmp/prov-va/tmp/config.sh",..
```

6.2 Graph Representation

A provenance graph that conforms to the model described in Section 4 was generated from the collected traces. The integrity measurement log provides integrity values of the components on the platform and therefore enables us to create instances of *IntegrityValue* and specify the *integrityOf* property. The trace generated by *strace* provides information about relationships among the programs executed and the data they use.

A slice of the resulting graph is shown in Figure 7. The graph shows how programs and data used in various processes that occurred on a VA are related and how IMA provides assertions about integrity values.

6.3 Verification

We demonstrate some of the trust properties that can be verified for VAs.

Verifying Package Authenticity: can be determined in two ways: check the integrity values recorded for each of the"*.rpm*" packages or determine whether they were obtained from a trusted repository. Listing 1.7 and 1.9 shows two queries and the results obtained are shown in Listing 1.8 and 1.10, respectively.

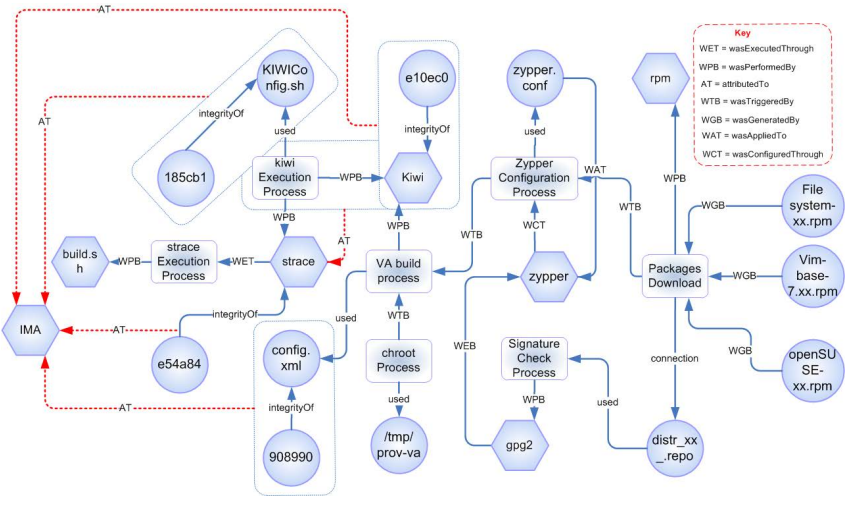


Fig. 7. A simplified provenance graph — showing only a small number of integrity values. The full graph includes a link between an integrity value for each of the artifacts and programs to the IMA.

Listing 1.7. Query that returns the integrity of all the packages installed

```
SELECT    {?pkg ?iv }
WHERE     {?pkg  :wasGeneratedBy ?x .
     ?iv      :integrityOf     ?pkg .
     FILTER regex(?x,"rpm")}
```

Listing 1.8. Results of package source query

```
Artifact                    IntegrityValue
"filesystem-xx-.i586.rpm"     "b0d515"
"vim-base-7.xx.i586.rpm"      "bc4d2a"
"openSUSE-xx-.i586.rpm"       "3b2310"
```

Listing 1.9. Query to return the mapping of the packages to the source

```
SELECT    {?pkg  ?y }
WHERE     {?x     :connection      ?y .
     ?xrdf:type   :Download .
     ?pkg     :wasGeneratedBy ?x .}
```

Listing 1.10. Results of package source query

```
Artifact                    Connection
"filesystem-xx-.i586.rpm"     opensuse.org_distr_xx_.repo
"vim-base-7.xx.i586.rpm"      opensuse.org_distr_xx_.repo
"openSUSE-xx-.i586.rpm"       opensuse.org_distr_xx_.repo
```

Verifying Configurations Applied: can be verified by checking the integrity values of the configuration settings that have a *wasAppliedTo* relationship with programs. Listing 1.11 shows the query performed.

Listing 1.11. Example query to check configurations

```
SELECT    { ?y ?x ?z}
WHERE       { ?x :integrityOf    ?y .
     ?p rdf:type :ConfigurationProcess .
     ?p :used     ?y .
     ?z :wasConfiguredThrough ?p .
     ?y :wasAppliedTo ?z}
```

Listing 1.12. Results of the checking program configurations

```
ConfigurationSettings    IntegrityValue Program
"zypper.conf"          "bddd59"      "zypper"
"KIWIConfig.sh"        "185cb1"      "Kiwi"
```

Verifying Startup Scripts : is accomplished by checking the integrity of the programs or artifacts that have been placed in a certain location. Listing 1.13

Listing 1.13. Example query to determine the scripts that will be executed

```
SELECT    {?x ?y }
WHERE     {?x rdf:type :artifact .
          ?y :integrityOf ?x
          FILTER regex(?x, "^init") }
```

Listing 1.14. Results of checking scripts copied

```
Artifact IntegrityValue
"config.sh" "3de324"
"image.sh" "359aa3"
```

7 Discussion

7.1 Interoperability and Extensibility

Our model can be extended with semantics useful for a given application domain by defining new concepts or extending existing concepts. The new concepts can then be linked to concepts that exist in the model or to other new ones by defining or using existing properties. For example, entities in the TCG schemas can be mapped to either programs or artifacts to take advantage of the interrelations among components and thus enable a more comprehensive verification of platform configurations.

7.2 Collecting and Securing the Evidence

One key issue with the use of provenance in verifying trustworthiness is establishing the trustworthiness of the provenance itself. Considerable effort [12,11] has been directed towards this end and it is not our intention to provide a solution for this problem. Instead, we have assumed that this information is secured and concentrated on developing a model that enables one to use this information in trustworthiness verification. The evidence can further be tagged with trust values to indicate the belief that the assertor has in the assertions [11], allowing quantitative measurement of trust.

7.3 Assumption on Infrastructure

In this paper, we consider trust properties that can be established through evidence obtained from the build platform. There are other aspects that could affect the behaviour of a virtual appliance when launched. For example, two identically configured VAs could behave differently if the runtime parameters passed from the hypervisor are different. We assume that the hypervisor would launch all VAs with identical parameters. We intend to investigate how such parameters could affect the behaviour as part of our future work.

8 Conclusions and Future Work

The nature of evidence for use in trust evaluation and how it can be represented and used is still an open problem. Existing mechanisms such as the TCG integrity schema are limited to a specific aspect of a system's operation (e.g. chain of program execution). We have proposed a provenance-based model in which evidence is represented as a provenance graph which captures activities that occur on a system. This model specifies relationships among system components and data to enable evaluation against certain trust properties. Our application to virtual platforms for use in a trust domain demonstrate that the approach enables verification of more comprehensive properties. The model will be incorporated as part of the trust domain framework.

Acknowledgements. The work described here was supported by the Trust Domains project funded by UK TSB and EPSRC, reference TS/I002634/1. We thank David Power and the anonymous reviewers for their insightful comments.

References

1. Cooper, A.: Towards a Trusted Grid Architecture. PhD thesis, Oxford University (2008)
2. Schmidt, A.U., Leicher, A., Shah, Y., Cha, I.: Tree-formed verification data for trusted platforms. CoRR, abs/1007.0642 (2010)

3. Kandukuri, B.R., Paturi, V.R., Rakshit, A.: Cloud security issues. In: IEEE International Conference on Services Computing, SCC 2009, pp. 517–520 (September 2009)
4. Namiluko, C., Huh, J.H., Martin, A.: Verifying Trustworthiness of Virtual Appliances in Collaborative Environments. In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) Trust 2011. LNCS, vol. 6740, pp. 1–15. Springer, Heidelberg (2011)
5. Grawrock, D.: Dynamics of a Trusted Platform: A Building Block Approach. Intel Press (2009)
6. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. Technical report, World Wide Web Consortium (January 2008)
7. Lyle, J., Martin, A.: Trusted computing and provenance: better together. In: Proceedings of the 2nd Conference on Theory and Practice of Provenance, TAPP 2010, p. 1. USENIX Association, Berkeley (2010)
8. Wei, J., Zhang, X., Ammons, G., Bala, V., Ning, P.: Managing security of virtual machine images in a cloud environment. In: Proceedings of the 2009 ACM Workshop on Cloud Computing Security, CCSW 2009, pp. 91–96. ACM, New York (2009)
9. Moreau, L., Freire, J., Futrelle, J., McGrath, R., Myers, J., Paulson, P.: The open provenance model (December 2007)
10. Santos, N., Gummadi, K.P., Rodrigues, R.: Towards trusted cloud computing. In: Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, HotCloud 2009. USENIX Association, Berkeley (2009)
11. Hartig, O.: Querying Trust in RDF Data with tSPARQL. In: Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 5–20. Springer, Heidelberg (2009), doi:10.1007/978-3-642-02121-3_5
12. Groth, P., Moreau, L.: Recording process documentation for provenance. IEEE Transactions on Parallel and Distributed Systems 20(9), 1246–1259 (2009)
13. Groth, P., Moreau, L.: Representing distributed systems using the open provenance model. Future Generation Computer Systems 27(6), 757–765 (2011)
14. Presti, S.L.: A tree of trust rooted in extended trusted computing. In: Proceedings of the Second Conference on Advances in Computer Security and Forensics Programme (ACSF), pp. 13–20 (2007)