Access Control Based on Code Identity for Open Distributed Systems

Andrew Cirillo and James Riely*

CTI, DePaul University
{acirillo,jriely}@cs.depaul.edu

Abstract. In computing systems, *trust* is an expectation on the dynamic behavior of an agent; *static analysis* is a collection of techniques for establishing static bounds on the dynamic behavior of an agent. We study the relationship between code identity, static analysis and trust in open distributed systems. Our primary result is a robust safety theorem expressed in terms of a distributed higher-order pi-calculus with code identity and a primitive for remote attestation; types in the language make use of a rich specification language for access control policies.

Keywords: Trusted Computing, Remote Attestation, Access Control, Authorization Logic, Compound Principals, Higher-Order Pi Calculus, Typing.

1 Introduction

Trust is an important concept in computer security. One may think of trust as an expectation on the *behavior* of some agent. We say that an agent is *trusted* if the achievement of a security goal is dependent on the agent behaving in the expected way. An agent is *trustworthy* if it behaves in the expected way in all circumstances.

An effective way to determine that an agent is trustworthy is to establish bounds on its behavior through static analysis of its software components. Many important security-related behavioral properties can be usefully established statically, including memory and type safety, non-interference, compliance with mandatory and discretionary access control policies and adherence to an ad-hoc logical policy specification.

An *open* system is one in which software components are under the control of multiple parties whose interests do not necessarily coincide. The use of static analysis in these systems is more complicated than in *closed* systems, where all components are under the control of a single party.

To discuss the issues involved, we find it useful to distinguish software components according to their relative *roles*. Given a particular unit of code and a statically derivable property, we distinguish four primary roles: the *producer* is the original author of the code; a *host* is a system that executes, or is considering executing, the code; a *certifier* is a third party capable of performing an analysis directly on the code that determines whether the property holds; and a *relying party* is the entity whose safe operation depends on the property holding for the code.

* This work was supported by the National Science Foundation under Grant No. 0347542.

When code is distributed in a compiled format, it may be the case that only the producer, who has the original source, is able to tractably certify many important properties. A host for the compiled code, if it is a relying party, may not able to establish the properties it needs.

This problem is well studied, and at least two solutions have been developed. By distributing the executable as intermediate-level *bytecode*, the analysis may be made tractable; in this case many useful analyses may remain intractable, or at least impractical. With *proof-carrying code* [1] the producer uses a certifying compiler to generate a proof of the desired property that can be checked efficiently by the host; this allows a greater range of analyses, but with the limitation that properties have to be agreed upon in advance.

A second issue arises when the relying party and host systems are physically distinct. For example, a server may hold sensitive data that it is only willing to release to remote clients that are known to be running certifiably safe code. The certification could be done by the client, but on what grounds can the server trust the results? The certification can instead be done by the server, but only if it can authenticate the code running on the client.

In conventional authentication protocols, remote parties authenticate themselves by demonstrating knowledge of a secret. When executables are distributed over public channels, however, embedded secrets are vulnerable to extraction and misuse by attackers so code cannot in general be relied upon to authenticate itself. This problem is addressed in part by *trusted computing*, where a trusted host authenticates the code it is running, and when necessary attests to the identity of the code to remote parties.

Remote code authentication, or *attestation*, is based on measurements of static executables. Therefore, trusted computing platforms only attest to initial states of processes. This makes static analysis particularly important for reasoning in systems using attestation. Code identity is a degenerate example of a static property; more abstract properties can be defined as sets of executables that satisfy the property. Knowing that the executable running on a host satisfies a certain property may allow a relying party to determine something about the dynamic state of the host.

Even weak static properties may be useful in validating trust. For example, knowing that a server has the latest patches applied may ease the mind of an e-commerce client. Similarly, a bounded model checker or test suite may give some assurance of memory safety without proving absolute trustworthiness.

For concreteness, we concentrate here on access control properties established via a type system, leaving the general case to future work. This focus allows us to establish absolute guarantees of trustworthiness and thus to prove a robust safety theorem. We do so in the context of a higher-order π-calculus enhanced with process identity and primitive operations for remote attestation.

The contributions of this paper are twofold. First, we illustrate how the trusted computing paradigm can be used to enforce an access control model based on static properties of code. Second, we demonstrate the importance of higher-order languages in studying policies and protocols that make use of remote attestation.

Organization. In the remainder of this introduction, we provide some intuitions about our formalism and results. In Section 2 we present the syntax and operational semantics of our language. Detailed examples follow in Section 3. Section 4 summarizes the type system and the main safety theorem; details are elided here for lack of space. Related work is discussed in Section 5.

Background: Remote Attestation. Remote attestation is a technique that allows processes to prove their identity to remote systems through the use of a trusted third party that is physically collocated with the process. In the Trusted Computing Group (TCG) specification [2] this comes in the form of a Trusted Platform Module (TPM) – an embedded co-processor that has the ability to measure the integrity of the boot sequence and securely store cryptographic keys. Each TPM is created with a unique keypair and a certificate from a trusted certificate authority.

The TPM serves as the root of trust for a Trusted Software Stack (TSS) [3], which in turn serves a trusted operating system which hosts user programs. As the software stack progresses, a measurement (cryptographic hash) of the next item to be loaded is placed in a secure register before it executes. Upon request, the TPM will produce a signature for the contents of the secure register bank using a private key. An *attestation* is a list of measurements plus a payload, signed by a TPM key.

Measurements of program executables, in this case, serve as a form of code identity. Modifying an executable changes its measurement, so attestation effectively identifies the remote process, and also demonstrates that the software running on the remote system has not been compromised.

We do not, in this paper, attempt to model the underlying protocol of remote attestation using explicit cryptographic primitives [4] nor do we attempt to translate our calculus into a lower level calculus with cryptographic primitives [5]. Instead, we take it for granted that the following capabilities are available and treat attestation as a primitive operation. We assume that executables can be measured in a globally consistent fashion (e.g., using an SHA-1 hash), and that the keys embedded in each TPM are issued by a globally trusted (and trustworthy) certificate authority. We also assume that, when multitasking, trustworthy operating systems enforce strong memory isolation between processes. Attestation protocols [6] are designed to be anonymous, so we do not assume any capability for distinguishing between different instances of a program nor do we assume that any information is available regarding the physical location of a process.[1]

Access Control with Remote Attestation. Remote attestation enables a model of access control in which executables, as identified by their cryptographic hashes, assume the role of principal. In its most basic form, it allows an attesting system to demonstrate to remote parties exactly what executables it has loaded. The remote party may exercise a simple form of access control by choosing to continue interacting with the attesting system only if all of its loaded executables are known and trusted. For example, an online media server may refuse access to clients not running its digital rights management (DRM) software.

While this simple approach may be sufficient in a limited context where only a small number of well-known executables need be trusted, such as in the proprietary DRM

[1] While attestations are anonymous in the sense that an individual user or machine cannot be identified, the recipient does get precise information about the software and hardware running on the attesting system that could be considered sensitive. Sadeghi and Stüble [7] cite this as a shortcoming of the TCG specification in their argument for property-based attestation.

example above, it is low-level, inefficient and inflexible. A common criticism [8] of trusted computing cautions that this lack of flexibility could be used by industry leaders to lock out open-source software and products from smaller competitors.

A more robust design is necessary to broaden the applicability of trusted computing, and indeed a number of extensions to the existing specification have already been proposed [7,9,10].

Overview of Our Solution. Modeling systems that operate on static units of executable code is a suitable task for a higher-order π-calculus [11,12,13], where processes can be abstracted and treated as data. Thus, we develop a higher-order π-calculus, dubbed HOπ-rat, enhanced with process identity and primitives for creating and using attestations. Process identity is implemented in the form of *configurations*, which are located processes where location is a representation of the identity (measurement) of the software stack that spawned the process.

Access control in HOπ-rat is based on a notion of principal that is tied to code identity. Static properties of code also play a role. We model these qualities as membership in a *security class*. Security classes are the basis for our access control policies, and a single executable may belong to multiple security classes. Complex principals are specified by a language of compound principals that includes both primitive identities and security classes.

There are two aspects to access control policy. First, read and write authorizations are specified explicitly in type annotations on channels in the form of expressions in the language of compound principals. Second, the sort of trust that a process places in particular identities is represented as a mapping of identities to security classes.

Our security classes are flexible, and can accommodate a wide range of security expectations, however one expectation is distinguished: each participant in a trusted software stack must maintain the expectations of the trusted system as a whole. In particular, they must not do anything to compromise the integrity of an attestation, and they must not leak secret data on insecure channels. We designate this behavior with the security class cert. We also develop a notion of *robust safety* and present a sketch of a type system that ensures robust safety in the presence of arbitrary attackers. We discriminate between typechecked and non-typechecked identities via membership in cert, and refer to typechecked processes as *certified*.

2 The Language

In this section we describe the syntax and operational semantics of the HOπ-rat calculus. We first define a sub-calculus of compound principals that will serve as the basis for access control in our system. We then define the syntax of terms, types, processes and configurations, followed by the operational semantics.

2.1 A Calculus of Compound Principals

To support the creation of sophisticated access control policies, we develop a calculus of compound principals in the style of Abadi et al. [14]. Primitive principals include identities and classes (including the distinguished class cert) drawn from an infinite set

($\mathcal{N}$) of atomic principal names. The principal constant 0 represents the inert process – always trustworthy by virtue of its inertness – and any represents an arbitrary process.

Compound principals are constructed using conjunction ($\wedge$), disjunction ($\vee$) and quoting ($|$) operators. Of these, quoting is essential because it is used to represent one process running in the context of another. For example, the principal $tss|myos|widget$ might represent a user application running on an operating system running on a trusted software stack. The other combinators are provided only to add expressiveness to the policy language.

A *policy environment* (Σ) maps identities to classes. For example, $\Sigma(a,\alpha)$ indicates that a is a member of α. Class membership is many-to-many; an identity may be a member of multiple classes, and a class may have multiple members. We write $a \Rightarrow \alpha$ for a policy environment consisting of a single pair.

PRINCIPALS AND POLICY ENVIRONMENTS

$a,b,c \in \mathcal{N}_{id} \quad \alpha - \omega \in \mathcal{N}_{cls}$ (including cert)	*identities/classes*
$A,B,C ::= \mathsf{any} \mid 0 \mid a \mid \alpha \mid A \wedge B \mid A \vee B \mid A|B$	*principals*
$\Sigma,\Phi \subseteq \mathcal{N}_{id} \times \mathcal{N}_{cls}$	*policy environment*

We define a partial order ($\Rightarrow$), ranking principals in terms of trustedness. When $A \Rightarrow B$, A is trusted at least as much as B. Derivations are defined in terms of a policy environment so that $\Sigma \vdash A \Rightarrow B$ if $\Sigma(A,B)$, or $A = 0$, or $B = \mathsf{any}$, so that $\wedge,\vee$ are commutative, associative, idempotent, absorptive and distribute over each other, so that $|$ is monotone and idempotent, and so that $\Rightarrow$ is reflexive, transitive and antisymmetric. Thus defined, $\Rightarrow$ forms a distributive lattice with $\wedge,\vee$ as meet and join operators, and $\mathsf{any}, 0$ as top and bottom elements. If Σ and Φ are policy environments, we write $\Sigma \vdash \Phi$ if for every a,α such that $\Phi(a,\alpha)$, we have that $\Sigma \vdash a \Rightarrow \alpha$.

Our treatment of compound principals builds on existing work [14,15]. In the interest of minimality, we use only a calculus of principals and do not incorporate a full modal authorization logic, which would include a "says" construct. Existing techniques [16,17,18] for using authorization logics in π-calculi could be applied here as well.

2.2 Syntax

In addition to principals, the main syntactic categories of HOπ-rat are terms, types, processes and configurations. As usual in π, we assume an infinite set ($\mathcal{N}$) of names, but we distinguish channels (n,m) from variables (x,y,z). We use a *local* syntax [19,20,21] in the sense that only output capabilities may be communicated as it is syntactically disallowed to read from a variable.

TERMS

$n,m \in \mathcal{N}_{ch} \quad x,y,z \in \mathcal{N}_{var}$	*channels/variables*
$M,N ::= n \mid x \mid \mathtt{unit} \mid (x:T)P \mid (M,N) \mid [M:T] \mid \{M:T \text{ @ } A\}_*$	*terms*

Terms include channel names, variables, a unit term and process abstractions from higher-order π, pairs, and two novel constructs. The term $[M:T]$, where M is a process abstraction and T is an abstraction type, represents an executable. We assume that the identity of an executable can be taken directly using a well-known measurement algorithm, which we represent as a function (#) taking executable terms to primitive identities. Since otherwise trustworthy programs can sometimes be coerced to misbehave if they are initialized incorrectly, executables include a type annotation to ensure that the identity function takes the type of the program arguments into account.

The term $\{M : T \text{ @ } A\}_*$ represents an attestation – the payload M tagged with type T and the principal A, where A stands for a list of the measurements of the executables that were running when the attestation was requested.

TYPES

$$S,T ::= \mathsf{Ch}\langle A,B\rangle(T) \mid \mathsf{Wr}\langle A,B\rangle(T) \mid \mathsf{Unit} \mid T \to \langle A\rangle\mathsf{Proc} \mid S \times T \mid \mathsf{Tnt} \mid \mathsf{Un} \mid \mathsf{Prv} \mid \mathsf{Pub}$$

Types include constructs for read/write and write-only channels, unit, abstractions, pairs and four top types. The unit and pair types are standard; we discuss the others below.

Channel types include annotations for specifying policy. For example, the type $\mathsf{Ch}\langle A,B\rangle(T)$ is given to channels that communicate values of type T, and may be used for input by processes authorized at principal B with the expectation that it will only be used for output by processes authorized at principal A. As in Sangiorgi's localized pi [19], we syntactically restrict input to channels, disallowing input on variables. Therefore, channel types may only be used with names. Write types are similar, but only allow output and therefore may be used to type variables.

The security annotations allow for fine-grained specifications of access control policy. For example, a channel annotated with type $\mathsf{Ch}\langle \alpha \wedge \beta, B\rangle(T)$ can only be written on by processes that are members of both α and β. Conversely, $\mathsf{Ch}\langle \alpha \vee \beta, B\rangle(T)$ requires membership in either α or β. Other policies can place restrictions on the software stack, as in $\mathsf{Ch}\langle myos|\mathsf{any}, B\rangle(T)$, which permits any process running on the *myos* operating system.

Types for abstractions take the form $S \to \langle A\rangle\mathsf{Proc}$, where S is the type of the argument and A is a security annotation representing a principal that the process may expect to run at (discussed in Section 4). We sometimes write $S \to \mathsf{Proc}$ when the security annotation is not of interest.

Attestations and executables are typed at one of the four top types ($\mathsf{Tnt}, \mathsf{Un}, \mathsf{Prv}, \mathsf{Pub}$) which are used to classify data by secrecy and integrity properties. The top types are used in the kinding judgment mentioned in section 4.

PROCESSES AND CONFIGURATIONS

$$\begin{aligned}
P,Q ::=\; & 0 \mid n?N \mid \texttt{repeat}\; n?N \mid M!N \mid M\,N \mid \texttt{new}\; n:T;\; P \mid P \mid Q \\
& \mid \texttt{split}\;(x:S, y:T) = M;\; P \mid \texttt{let}\; x = \texttt{attest}(M:T);\; P \\
& \mid \texttt{check}\;\{x:T\} = M;\; P \mid \texttt{load}\; M\; \texttt{as}\; [T]\; N \\
& \mid \Sigma \mid \texttt{wr-scope}\; n\; \texttt{is}\; A \mid \texttt{rd-scope}\; M\; \texttt{is}\; A \\
& \mid \texttt{spoof}\; A; P \mid \texttt{let}\; \vec{x} = \texttt{fn}(M);\; P \\
G,H ::=\; & 0 \mid A[P] \mid G \mid H \mid \texttt{new}_A\; n:T;\; G
\end{aligned}$$

Processes include the usual constructs for HOπ: the inert process; input and replicated input; output; higher-order application, as in $M\,N$, which applies the argument N to the abstraction M; restriction; and parallel composition. The form $\mathtt{split}\ (x:S, y:T) = M;\ P$ is used to split a pair into its constituent parts.

The main security extensions are `attest`, `check` and `load`. The form $\mathtt{let}\ x = \mathtt{attest}(M:T);\ P$ represents a call to trusted hardware to create a new attested message with payload M and attested type T. The form $\mathtt{check}\ \{x:T\} = M;\ P$ tests and conditionally destructs an attestation. The form $\mathtt{load}\ M\ \mathtt{as}\ [T]\ N$ dynamically tests the identity and argument type of an executable prior to running it. The inclusion of Σ in the process language allows processes to carry knowledge about other processes at runtime. The expectations $\mathtt{wr\text{-}scope}\ n\ \mathtt{is}\ A$ and $\mathtt{rd\text{-}scope}\ M\ \mathtt{is}\ A$ are only used in the definition of runtime error and are discussed further below.

The final two forms are reserved for attackers, and therefore cannot appear in any well-typed term. The form $\mathtt{spoof}\ A; P$ allows the process to change its identity and the form $\mathtt{let}\ \vec{x} = \mathtt{fn}(M);\ P$ extracts the free names of a term.

Configurations (G, H) are composed of processes located at principals (e.g., $A[P]$). Our treatment of configurations is mostly standard for located π-calculi [22,23] with one exception: our locations expand as new code is loaded. For example, we use the compound principal $(a|b|c)$ to represent the sequence of a having loaded b having loaded c.

2.3 Operational Semantics

Evaluation is defined on configurations. We elide the structural equivalence rules which are mostly standard for located calculi [22] (for example "$A[P \mid Q] \equiv A[P] \mid A[Q]$"). The one novelty is the rule, "$\Sigma \mid \Phi \equiv \Sigma, \Phi$", which allows policy environments to be combined.

EVALUATION

$$\text{(R-COMM)}\quad A[n?M] \mid B[n!N] \longrightarrow A[M\,N] \qquad \text{(R-STRUC)}\quad \frac{G \equiv G' \quad H' \equiv H \quad G' \longrightarrow H'}{G \longrightarrow H}$$

$$\text{(R-APP)}\quad A[(x:T)P\,N] \longrightarrow A[P\{x := N\}] \qquad \text{(R-RES)}\quad \frac{G \longrightarrow G'}{\mathtt{new}\,n:T;\ G \longrightarrow \mathtt{new}\,n:T;\ G'}$$

$$\text{(R-ATT)}\quad A[\mathtt{let}\ x = \mathtt{attest}(M:T);\ P] \longrightarrow A[P\{x := \{M:T\ @\ A\}_*\}] \qquad \text{(R-PAR)}\quad \frac{G \longrightarrow G'}{G \mid H \longrightarrow G' \mid H}$$

$$\text{(R-SPLIT)}\quad A[\mathtt{split}\ (x:S, y:T) = (M,N);\ P] \longrightarrow A[P\{x := M\}\{y := N\}]$$

$$\text{(R-CAST)}\quad \frac{\Sigma \vdash S <: T \qquad \Sigma \vdash B \Rightarrow \mathsf{cert}}{A[\Sigma] \mid A[\mathtt{check}\ \{x:T\} = \{M:S\ @\ B\}_*;\ P] \longrightarrow A[\Sigma] \mid A[P\{x := M\}]}$$

$$\text{(R-CASTUN)}\quad A[\mathtt{check}\ \{x:\mathsf{Tnt}\} = \{M:S\ @\ B\}_*;\ P] \longrightarrow A[P\{x := M\}]$$

$$\text{(R-LOAD)}\quad \frac{\Sigma \vdash S <: T \rightarrow \langle B \rangle \mathsf{Proc} \qquad \Sigma \vdash a \Rightarrow \mathsf{cert}}{A[\Sigma] \mid A[\mathtt{load}\ [M:S]\ \mathtt{as}\ [T \rightarrow \langle B \rangle \mathsf{Proc}]\ N] \longrightarrow A[\Sigma] \mid (A|a)[M\,N]}\quad a = \#([M:S])$$

$$\text{(R-LOADUN)}\quad \frac{\vdash S <: T \rightarrow \langle B \rangle \mathsf{Proc} \qquad \vdash T <: \mathsf{Un} \qquad b = \#([M:S])}{A[\mathtt{load}\ [M:S]\ \mathtt{as}\ [T \rightarrow \langle B \rangle \mathsf{Proc}]\ N] \longrightarrow (A|b)[M\,N]}$$

$$\text{(R-SPOOF)}\quad A[\mathtt{spoof}\ B; P] \longrightarrow (A|B)[P]$$

$$\text{(R-PEEK)}\quad \frac{}{A[\mathtt{let}\ \vec{x} = \mathtt{fn}([M:T]);\ P] \longrightarrow A[P\{\vec{x} := fn(M)\}]}\ \text{if } |\vec{x}| = |fn(M)|$$

The rule for communication (R-COMM) passes a value along a channel in the standard way. When a value is communicated from one identity to another, the resulting process takes on the identity of the receiving process. The rule for splitting pairs (R-SPLIT) is standard.

In the rule for the creation of attestations (R-ATT) a term is tagged with a type and the pair is signed with the identity of the creating process. In the first rule for destruction (R-CAST), the identity of the generating process is recovered and tested against the local policy of the receiving process. The receiver must believe that the generating process is certified before it can trust the contents of the message. If the necessary facts are not present the destructor blocks, so for example these two configurations in parallel will reduce whereas the latter on its own would not.

$$A\big[b \Rightarrow \mathsf{cert}\big] \mid A\big[\texttt{check}\ \{x:T\} = \{N:T\ @\ b\}_*;\ P\big] \longrightarrow A\big[b \Rightarrow \mathsf{cert}\big] \mid A\big[P\{x:=N\}\big]$$

In order to safely unpack $\{N:T\ @\ B\}_*$ one must be able to establish that B is certified, that is that $B \Rightarrow \mathsf{cert}$ holds in the lattice of principals derived from the receiver's local policy. Note that from the idempotency and monotonicity of $\mid$ one can derive $a|b \Rightarrow \mathsf{cert}$ if $a \Rightarrow \mathsf{cert}$ and $b \Rightarrow \mathsf{cert}$. The principals used in attestations always have this form, so an attestation will be trusted if each of its component identities are certified. The receiving process need not know of all certified processes, only those with which it interacts, however a process may be unable to unpack a perfectly safe message if any identity in the sequence is unknown.

The second rule for destruction (R-CASTUN) allows a process to skip the dynamic checks if there are no type requirements for the extracted data (the type Tnt is at the top of the subtype hierarchy).

The rule for application (R-APP) converts an abstraction into a running process by substituting the argument for the bound variable. R-LOAD allows parent processes to run abstractions that they have received from untrusted sources after completing two dynamic checks. First, it tests the hash of M for certification. If M is known to be certified, then the type assertion can be trusted. Second, it tests that the asserted type is a subtype of the expected type. If both tests are successful, it extracts the enclosed abstraction and applies it to the argument.

As with attestations a second version (R-LOADUN) allows the dynamic checks to be skipped, in this case if the argument is of a safe type (i.e., contains no secrets). For example, suppose $b = \#([M:T])$. The following process located at A loads M.

$$A\big[b \Rightarrow \mathsf{cert}\big] \mid A\big[\texttt{load}\ [M:T]\ \texttt{as}\ [T]\ N\big] \longrightarrow A\big[b \Rightarrow \mathsf{cert}\big] \mid (A|b)\big[M\,N\big]$$

A's local mapping ($b \Rightarrow \mathsf{cert}$) indicates that $[M:T]$ is known to be certified, which enables the loading. Note that the residual is located at $A|b$.

The final two rules are reserved for uncertified systems and are necessary to model realistic attacks on higher-order code. R-SPOOF allows a process to impersonate an arbitrary principal as long as the root is preserved and R-PEEK allows a process to extract the free names of a higher-order term. Spying on, or "debugging," a child process can be modeled using a combination of these operations as follows: the attacker first extracts the free names of an executable, then builds a new executable identical to the original except that all bound names are replaced with names in the attacker's scope, and finally loads the modified executable and spoofs the identity of the original process.

3 Examples

In this section we illustrate the use of HOπ-rat in two detailed examples. Throughout this section we use the following notational conveniences: we elide trivial type annotations, we abbreviate $\texttt{load}\, M\ \texttt{as}\ [\mathsf{Un} \to \langle 0 \rangle \mathsf{Proc}]\, N$ as $\texttt{load}\, M\, N$, and we abbreviate $(x : \mathsf{Unit})P$ as $()P$ when $x \notin fn(P)$.

3.1 Example: A Trusted Software Stack

Our first example shows how the integrity of the software stack can be preserved in a trusted system, from the booting of the operating system to the execution of a user application. We start with a simple computer system composed of a BIOS (*BIOS*), disk drive (*DSKDRV*), user interface (*UI*) and operating system (*OS*). The first three components are loaded by hardware, thus they are represented as pre-existing processes. The operating system, however, must be booted from code stored on the disk drive.

We assume that the disk drive is untrusted. (Unencrypted storage devices are easily tampered with while the computer is switched off, so anything loaded from the disk drive must be treated as if it came from a public source.) The process representing the drive listens for file requests on a series of channels, one for each file, and responds by writing the file on the request channel. Some of these files will be executable programs; in particular, a request on the distinguished channel *mbr* (for *master boot record*) will return the operating system kernel code.

$$DSKDRV \triangleq \texttt{repeat}\, mbr?(x)x!OS \mid \texttt{repeat}\, f_i?(x)x!FILE_i \mid \ldots$$

The BIOS is responsible for locating and loading the operating system, which it does by sending a request on *mbr* and loading the returned executable. The BIOS does not need to verify the safety or identity of the executable because the load command stores the hash of the loaded program in the PCR, ensuring that it is reflected in the identity of the resulting process.

$$BIOS \triangleq \texttt{new}\, n;\ mbr!n \mid n?(y)\texttt{load}\, y\, \texttt{unit}$$

Let $dskdrv = \#([()DISKDRV])$, $bios = \#([()BIOS])$, and $os = \#([()OS])$. At startup the BIOS process will be located at *bios* and the disk process at *dskdrv*. The boot sequence proceeds as follows.

BOOTING WITH INTEGRITY

$$
\begin{aligned}
& bios\big[\texttt{new}\, n;\ mbr!n \mid n?(y)\texttt{load}\, y\, \texttt{unit}\big] \mid dskdrv\big[\texttt{repeat}\, mbr?(x)x![()OS] \mid \ldots\big] \\
\longrightarrow^4\ & bios\big[\texttt{load}\, [()OS]\, \texttt{unit}\big] \mid dskdrv\big[\texttt{repeat}\, mbr?(x)x![()OS] \mid \ldots\big] \\
\longrightarrow^2\ & (bios|os)\big[OS\big] \mid dskdrv\big[\texttt{repeat}\, mbr?(x)x![()OS] \mid \ldots\big]
\end{aligned}
$$

By the end of the boot process, the operating system code is running at the identity $bios|os$. The BIOS code has terminated, but its identity is reflected in the identity of the operating system process. This ensures that a malicious BIOS cannot compromise or impersonate a trusted operating system without detection.

Note that no access control checks are required for the boot process. We consider it to be perfectly acceptable for a trusted system to load untrusted code as long as the identity of that code is recorded. This distinguishes this boot sequence from a *secure boot*, which only executes trusted code.

Once loaded, the operating system code enters a loop listening for requests to start user programs. Requests come in the form of a channel name that corresponds to a file on disk, and an argument term. The operating system fetches the corresponding file from the disk drive and loads it, passing it the argument term.

$$OS \triangleq \texttt{repeat}\, req?(x)\texttt{split}\ (f, arg) = x;\ \texttt{new}\, n;\ (f!n \mid n?(y)\texttt{load}\, y\, arg)$$

The type of the argument term is not checked. If the executable were initialized with an argument of the wrong type it could cause the security of the resulting process to fail, therefore the evaluation rule (R-LOADUN) requires that the executable be annotated to accept arguments of type Un. Any certified executable with such an annotation will have been proven to operate safely with arbitrary arguments.

Now we can consider how the system responds to a user request to run a program. Let *ui* represent part of the user interface hardware (keyboard, mouse, etc.) for some system, and assume that the user has indicated a request to load the program *PROG* by keying in "prog *args*" to the interface.

Loading A User Program

$$\begin{aligned}
& dskdrv\big[\ldots \mid \texttt{repeat}\, prog?(x)x![(z)PROG] \mid \ldots\big] \\
& \mid (bios|os)\big[\texttt{repeat}\, req?(x)\texttt{split}\ (f, arg) = x;\ \texttt{new}\, n;\ (f!n \mid n?(y)\texttt{load}\, y\, arg)\big] \\
& \mid ui\big[req!(prog, args)\big] \\
\longrightarrow^4\ & dskdrv\big[\ldots \mid \texttt{repeat}\, prog?(x)x![(z)PROG] \mid \ldots\big] \\
& \mid (bios|os)\big[\texttt{repeat}\, req?(x)\texttt{split}\ (f, arg) = x;\ \texttt{new}\, n;\ (f!n \mid n?(y)\texttt{load}\, y\, arg)\big] \\
& \mid (bios|os)\big[\texttt{load}\ [(z)PROG]\ args\big] \\
\longrightarrow^2\ & dskdrv\big[\ldots \mid \texttt{repeat}\, prog?(x)x![(z)PROG] \mid \ldots\big] \\
& \mid (bios|os)\big[\texttt{repeat}\, req?(x)\texttt{split}\ (f, arg) = x;\ \texttt{new}\, n;\ (f!n \mid n?(y)\texttt{load}\, y\, arg)\big] \\
& \mid (bios|os|prog)\big[PROG\{z := args\}\big]
\end{aligned}$$

After several reduction steps, the user program (*PROG*) is running at the identity *bios*|*os*|*prog*, and the operating system is back in its original state, awaiting a new command.

A user program can also load another user program through the operating system functionality. The new identity of this program will be *bios*|*os*|*newprog*; it does not reflect the identity of the calling program as it would if the calling program had invoked the load command directly. The operating system loop only loads programs that are expecting arbitrary arguments, so there is no chance that a malicious program can use this functionality to misconfigure a trusted program while excluding its own measurement from the identity sequence.

This illustrates an important difference between stand-alone executables started through operating system functionality, as in the example above, and dynamically loaded modules, such as shared libraries and browser plugins. In the former case the operating system is solely responsible for the safe initialization of the code; in the latter, the calling process is relied upon to initialize the new module correctly, therefore its identity is reflected in the identity of the resulting process.

3.2 Example: Secure E-Commerce

In this example, remote attestation is used to facilitate secure communication between a vendor and customer. Each party has different security requirements. In order to complete the transaction the customer has to provide sensitive personal information – a credit card number and delivery address – and therefore requires that the vendor be secure and comply with an electronic privacy policy.

On the other side, because the vendor may have to cover the cost of fraudulent charges, it has an interest in ensuring that the request is coming from an actual user, and not a trojan horse or virus running on the customer's machine. They can accomplish this by requiring that the request come from an actual web browser (as opposed to a script, or other program) and that the browser be free from security holes.

The two main parties are the customer (*cust*) and vendor (*vend*) executables; but there are also the customer (*c_host*) and vendor (*v_host*) hosts. We represent the requirements that the customer has of the vendor with the security class *ok_vend*, and that the vendor has of the customer with *ok_cust*. These properties are established by two independent certifiers, *vendcc* and *custcc*.

The code for the customer certifier (*custcc*) is shown below, the vendor certifier is similar. It listens for requests on a well-known public channel (*getCustIsOk*), and responds with a certificate mapping the *cust* identity to the *ok_cust* security class. Recall that policy environments are part of the process language, so we communicate them as thunked processes. A certificate therefore has the form of a thunked policy environment wrapped in an attestation.

CUSTOMER CERTIFIER

```
$(\ldots|custcc)\big[$ repeat $getCustIsOk?(c)$
            let $msg = $ attest$(()\#(cust) \Rightarrow ok\_cust : \mathsf{Unit} \rightarrow \langle\mathsf{cert}\rangle\mathsf{Proc})$; $c!msg\,\big]$
```

The location of *custcc* is not important. It is the vendor process that requires the customer certifier, so they could be running on the same host, however the use of an attestation to sign the certificate means that the processes could just as easily be distributed. Trust is placed in the *program* doing the certification, not the physical node running it, so any node equipped with a TPM running the correct software – even the customer node itself – can host a certifier process.

At the start of the protocol, *cust* (1) trusts only the vendor certifier. It first consults a trusted certifier (2-3) and obtains a certificate listing some trustworthy vendors; *vend* (11-13) does the same but for trustworthy customers. The customer then initiates the protocol by creating (5) a partially secure (only the customer can read, but anyone can write) callback channel, wrapping it (6) in an attestation and forwarding it (7) to the vendor on a well-known public channel. At this point the attested message will have the form $\{cch : \mathsf{Wr}\langle\mathsf{any}, ok\_cust\rangle(\mathsf{Tnt})\ @\ c\_host|cust\}_*$. After receiving the message, the vendor performs a dynamic check (15) to ensure that the message is from a trusted source, and that the contents are of the expected type. Succeeding at that, it continues by creating its own secure callback (16), wrapping it in an attestation (17) and sending it back to the customer (18). At this point the parties have established bidirectional secure communications, and the customer data can be sent (10) safely with all security requirements met.

Note that in order for the dynamic checks (3,9,15,13) to pass, the process must explicitly trust the attestors. The trust required to allow the first checks (3,13) to pass is already hard-coded (1,11) in the executables. The trust required for the other checks (9,15) are acquired at runtime from the trusted certifiers.

CUSTOMER AND VENDOR EXECUTABLES

```
$(c\_host|cust)[$
 1  $vendcc \Rightarrow \mathsf{cert} \mid v\_host \Rightarrow \mathsf{cert} \mid$
 2  $\mathtt{new}\, c;\ getVendIsOk!c \mid c?(x:\mathsf{Un})$
 3    $\mathtt{check}\ \{y:\mathsf{Unit} \rightarrow \langle\mathsf{cert}\rangle\mathsf{Proc}\} = x;\ x\,\mathtt{unit} \mid$
 4  $\mathtt{new}\, address, credit\_card : \mathsf{Ch}\langle\mathsf{cert}, ok\_cust\rangle(\mathsf{Prv});$
 5  $\mathtt{new}\, cch : \mathsf{Ch}\langle\mathsf{any}, ok\_cust\rangle(\mathsf{Tnt});$
 6  $\mathtt{let}\, amsg = \mathtt{attest}(cch : \mathsf{Wr}\langle\mathsf{any}, ok\_cust\rangle(\mathsf{Tnt}));$
 7  $vpub!amsg \mid$
 8  $cch?(x:\mathsf{Tnt})$
 9    $\mathtt{check}\ \{y:\mathsf{Wr}\langle ok\_cust, ok\_vend\rangle(\mathsf{Prv})\} = x;$
10    $y!(address, credit\_card)\ ]$

$(v\_host|vend)[$
11  $custcc \Rightarrow \mathsf{cert} \mid c\_host \Rightarrow \mathsf{cert} \mid$
12  $\mathtt{new}\, c;\ getCustIsOk!c \mid c?(x:\mathsf{Un})$
13    $\mathtt{check}\ \{y:\mathsf{Unit} \rightarrow \langle\mathsf{cert}\rangle\mathsf{Proc}\} = x;\ x\,\mathtt{unit} \mid$
14  $vpub?(x:\mathsf{Un})$
15    $\mathtt{check}\ \{y:\mathsf{Wr}\langle\mathsf{any}, ok\_cust\rangle(\mathsf{Tnt})\} = x;$
16    $\mathtt{new}\, vch : \mathsf{Ch}\langle ok\_cust, ok\_vend\rangle(\mathsf{Prv});$
17    $\mathtt{let}\, vmsg = \mathtt{attest}(vch : \mathsf{Wr}\langle ok\_cust, ok\_vend\rangle(\mathsf{Prv}));$
18    $y!vmsg$
19    $vch?(data:\mathsf{Prv})(\ldots continue\ processing\ transaction \ldots)\ ]$
```

4 A Type System for Certified Processes

We have developed a type system that ensures that typed processes meet the behavioral requirements for certified processes, even in the presence of arbitrary attackers. For space reasons, most of the details are elided.

We begin by formalizing the requirements as a definition of *robust safety*. Attackers come in two forms: as any software stack running on a system without a TPM, and as an untrusted process running on an otherwise trusted system. Our assumptions about attackers are as liberal as possible. The only requirements are that they be located at an uncertified identity, and that any attestations they possess must be acquired at runtime. In addition, we allow attackers to do the following: (1) if they are of the latter form, they may create attestations that extend the measurement sequence arbitrarily, provided that the measurements up to and including the untrusted process are accurate, (2) they may extract the contents of executables, including any embedded keys, and (3) they may peek at the memory of (i.e., debug) running child processes.

DEFINITION 1 (INITIAL ATTACKER). Let H be a configuration and Σ a policy environment. H is considered a Σ-*initial attacker* if it is of the form $A_1\left[P_1\right]\ldots A_n\left[P_n\right]$ where $(\forall i)\Sigma \nvdash A_i \Rightarrow \mathsf{cert}$, and it has no subterms of the form $\{M : T \ @ \ B\}_*$

Robust Safety. Safety is defined in terms of runtime error. The full system includes shape errors in addition to the access control errors presented here.

RUNTIME ERROR (PARTIAL)

$$\text{(E-WRSCP)}\ \frac{}{\Sigma \rhd A\left[\mathtt{wr\text{-}scope}\, n \,\mathtt{is}\, C\right] \mid B\left[n!N\right] \overset{error}{\longrightarrow}}\ \textit{if}\ \Sigma \vdash A \Rightarrow \mathsf{cert}\ \textit{and}\ \Sigma \nvdash B \Rightarrow C$$

$$\text{(E-RDSCP)}\ \frac{}{\Sigma \rhd A\left[\mathtt{rd\text{-}scope}\, n \,\mathtt{is}\, C\right] \mid B\left[n?N\right] \overset{error}{\longrightarrow}}\ \textit{if}\ \Sigma \vdash A \Rightarrow \mathsf{cert}\ \textit{and}\ \Sigma \nvdash B \Rightarrow C$$

A configuration is in error, for example, if a certified configuration is expecting the write scope of a channel to be restricted to one principal, and the channel is written on by a process located at another principal that does not carry that level of authorization in the lattice of principals.

Robust safety requires that no certified process can lead to a runtime error even in the presence of arbitrary attackers. It is defined relative to a policy environment, so it is perfectly reasonable to have policies that disagree on the safety of a given process. Our main result is that well-typed configurations are robustly safe.

DEFINITION 2 (ROBUST SAFETY). Let G be a configuration and G' a Σ-initial attacker. If $G \mid G' \longrightarrow^* H$ implies that $\Sigma \rhd H \overset{error}{\not\longrightarrow}$ for an arbitrary G' then we say that G is *robustly* Σ*-safe*.

THEOREM 3 (ROBUST SAFETY). Let G be a configuration, Σ a policy and Γ a global environment. If all of the the type assignments in Γ are of the form $\mathsf{Ch}\langle\mathsf{any},\mathsf{any}\rangle(\mathsf{Un})$, and $\Sigma;\Gamma \Vdash G$, then G is robustly Σ-safe.

Typing Rules. Types are constrained by kinding rules which prevent secret data from leaking to uncertified processes, or typed data from being read from an uncertified source. Subtyping allows integrity guarantees to be relaxed and write authorization requirements to be constrained. Our development of kinds and subtyping borrows heavily from Jeffrey and Gordon [24] and Haack and Jeffrey [25], and is similar to the system presented in [23].

The rules for terms and processes tag abstractions with the principal that it impersonates. For example, a process that uses a channel reserved for α will type as $\langle\alpha\rangle\mathsf{Proc}$, and one that uses both α and β channels will type as $\langle\alpha \wedge \beta\rangle\mathsf{Proc}$. If M is an abstraction that takes an argument of type T and makes use of α and β channels, it will type as $T \rightarrow \langle\alpha \wedge \beta\rangle\mathsf{Proc}$. Our technique for typing processes and process abstractions is similar to that of Yoshida and Hennessy [26], although our types are less precise than theirs in that we only record the authorizations required rather than the exact channels used.

Rules ensure that processes located at certified principals typecheck at a type compatible with that principal. For example, a process that types at $\langle\alpha\rangle\mathsf{Proc}$ can be located at a principal A only if $\Sigma \vdash A \Rightarrow \alpha$. There are, on the other hand, no constraints on locating processes at uncertified principals.

Consistency requirements for enforceable policies ensure that 1) only typechecked executables are assigned to class cert, and 2) typechecked executables that are assigned to cert are also assigned to other classes they require. For example, suppose M types at $T \to \langle\alpha \wedge \beta\rangle\mathsf{Proc}$. If a policy assigns $\#([M : T \to \langle\alpha \wedge \beta\rangle\mathsf{Proc}])$ to cert, then it must also assign it to α and β to be considered enforceable.

5 Related Work

This paper expands on our prior work [23] in two ways. First, the use of a higher-order calculus allows us to describe code distribution and loading. Second, the incorporation of security classes and a calculus of principals allow for rich specification of policy.

Abadi [27] outlines a broad range of trusted hardware applications that use remote attestation to convey trust assertions from one process to another. Our work can be seen as a detailed formal study of a specific kind of trust assertion, namely information about the type and access control policy for communicated code.

The NGSCB [28] remote attestation mechanism, and the TCG [29] hardware that underpins it, are more complex than the HOπ-rat remote attestation mechanism. We have omitted much of the complexity in order to focus on the core policy issues. For a logical description of NGSCB's mechanism, see [30]. For a concrete account of implementing NGSCB-like remote attestation on top of TCG hardware see [31].

Haldar, Chandra, and Franz [32,33] use a virtual machine to build a more flexible remote attestation mechanism on top of the primitive remote attestation mechanism that uses hashes of executables. Sadeghi and Stüble [7] observe that systems using remote attestation may be fragile, and discuss a range of options for implementing more flexible remote attestation mechanisms based upon system properties (left unspecified, as the focus is upon implementation strategies). Sandhu and Zhang [9] consider the use of remote attestation to protect disseminated information.

Our formal development builds upon existing work [34,24] with symmetric-key and asymmetric-key cryptographic primitives in pi-calculi. Notably, the kinding system is heavily influenced by the pattern-matching spi-calculus [25]. Our setting is quite different, however. In particular, processes establish their own secure channels and corresponding policies, as opposed to relying upon a mutually-trusted authority to distribute initial keys and policies. In addition, the access control policies used here are not immediately expressible in spi, since processes do not have associated identity. The techniques used to verify authenticity and other properties as in [35,36] should be applicable to HOπ-rat, though we make no attempt to address authenticity or replay attacks here. Finally, our primitive for checking attestations includes an implicit notion of *authorization*, which is made explicit in [25]. Scaling up to explicit authorizations would allow the possibility of enforcing policies that require multiple authorizations for certain actions.

Authorization based on code identity is also used by Wobber et al. in the context of the Singularity operating system [37], as well as in stack inspection [38] and other history-based access control policies [39]. Remote attestation can be used to implement similar policies in a distributed environment, but we leave this for future work.

The HOπ-rat type system allows executables to be typechecked independently and subsequently linked together. Separate compilation and linkability is not a new idea in programming languages, see, for example, [40], but is uncommon in spi-like calculi because there is usually a need to reliably distribute some shared secret or untainted data between separate processes in accordance with a type (policy). Recently Bugliesi, Focardi, and Maffei [41,42] have considered separate typechecking in the context of a spi-like calculus.

We assume that trusted hardware is trustworthy. For accounts of the difficulties involved in creating such trusted hardware, see [43,44] for an attacker's perspective and [45,46] for a defender's perspective. Irvine and Levin [47] provide a warning about placing too much trust in the integrity of COTS.

6 Conclusions

We defined a new extension to the higher-order π-calculus for analyzing protocols that rely on remote attestation. Our system extends our previous work [23] by incorporating higher-order processes and using a logic of principals to specify policy. This development allows parties to establish the identity and integrity of a remote process even if its executable has been exposed to attackers, but also allows us to expand the access control model from one based only on specific executables to one that incorporates abstract properties of code. This is an important advancement over existing capacities because these properties can include static analyses that establish bounds on the dynamic state of a remote host. We also provide a static analysis technique for ensuring robust safety in the presence of arbitrary attackers.

For future work, we are interested in internalizing program analysis, such as trusted compilers, typecheckers or code verifiers. This would allow us to model systems in which analysis tools are applied to programs at an enterprise boundary, then freely communicated and used within the enterprise without further analysis. We believe that such systems are very desirable, in that an enterprise may require that all code to be run in its systems must pass certain requirements. These requirements can be expressed as membership in a HOπ-rat security class. Analysis may be performed once, leading to a certificate (attestation) that the code belongs to the security class. The certificates may be communicated with the code, or independently, and verified through an efficient check of the hash of the code itself. We intend that these certificates be signed by the analysis tool itself, running on trusted hardware, rather than by an entity (such as a corporation) that vouches for the analysis. The use of hashes and rich policy specifications brings us close to being able to reason about such systems; HOπ-rat, as presented here, lacks only the ability to dynamically analyze abstractions.

References

1. Necula, G.C.: Proof-carrying code. In: POPL 1997 (1997)
2. Trusted Computing Group: TCG TPM Specification Version 1.2 (March 2006), `http://www.trustedcomputinggroup.org`

3. Trusted Computing Group: TCG Software Stack (TSS) Specification Version 1.2 (January 2006), `http://www.trustedcomputinggroup.org`
4. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Information and Computation 148(1) (1999)
5. Abadi, M., Fournet, C., Gonthier, G.: Authentication primitives and their compilation. In: POPL, pp. 302–315. ACM Press, New York, NY, USA (2000)
6. Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. In: CCS 2004: Proceedings of the 11th ACM conference on Computer and communications security, pp. 132–145. ACM Press, New York, NY, USA (2004)
7. Sadeghi, A.R., Stüble, C.: Property-based attestation for computing platforms: Caring about properties, not mechanisms. In: New Security Paradigms Workshop (2004)
8. Schoen, S.: Trusted Computing: Promise and Risk. Electronic Frontier Foundation (October 2003), `http://www.eff.org/Infrastructure/trusted_computing/20031001_tc.pdf`
9. Sandhu, R., Zhang, X.: Peer-to-peer access control architecture using trusted computing technology. In: SACMAT 2005: Proceedings of the tenth ACM symposium on Access control models and technologies, pp. 147–158. ACM Press, New York, NY, USA (2005)
10. Jaeger, T., Sailer, R., Shankar, U.: Prima: policy-reduced integrity measurement architecture. In: SACMAT 2006: Proceedings of the eleventh ACM symposium on Access control models and technologies, pp. 19–28. ACM Press, New York, NY, USA (2006)
11. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD thesis, University of Edinburgh (1993)
12. Thomsen, B.: Plain chocs: A second generation calculus for higher order processes. Acta Informatica 30(1), 1–59 (1993)
13. Milner, R.: Functions as processes. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 167–180. Springer, Heidelberg (1990)
14. Abadi, M., Burrows, M., Lampson, B., Plotkin, G.: A calculus for access control in distributed systems. ACM Trans. Program. Lang. Syst. 15(4), 706–734 (1993)
15. Abadi, M., Birrell, A., Wobber, T.: Access control in a world of software diversity. Tenth Workshop on Hot Topics in Operating Systems (HotOS X) (June 2005)
16. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization policies. In: ESOP, pp. 141–156 (2005)
17. Fournet, C., Gordon, A., Maffeis, S.: A type discipline for authorization in distributed systems. CSF 00, 31–48 (2007)
18. Cirillo, A., Jagadeesan, R., Pitcher, C., Riely, J.: Do As I SaY! Programmatic Access Control with Explicit Identities. CSF 0, 16–30 (2007)
19. Sangiorgi, D.: Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). Theoretical Computer Science 253, 311–350 (2001)
20. Yoshida, N.: Minimality and separation results on asynchronous mobile processes. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, p. 131. Springer, Heidelberg (1998)
21. Merro, M.: Locality in the pi-calculus and applications to distributed objects. PhD thesis, Ecole des Mines de Paris (October 2000)
22. Hennessy, M., Riely, J.: Resource access control in systems of mobile agents. Inf. Comput. 173(1), 82–120 (2002)
23. Pitcher, C., Riely, J.: Dynamic policy discovery with remote attestation. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, Springer, Heidelberg (2006)
24. Gordon, A.D., Jeffrey, A.S.A.: Types and effects for asymmetric cryptographic protocols. J. Computer Security 12(3/4) (2004)
25. Haack, C., Jeffrey, A.S.A.: Pattern-matching spi-calculus. In: Proc. IFIP WG 1.7 Workshop on Formal Aspects in Security and Trust (2004)

26. Yoshida, N., Hennessy, M.: Assigning types to processes. LICS 00, 334 (2000)
27. Abadi, M.: Trusted computing, trusted third parties, and verified communications. In: SEC 2004: 19th IFIP International Information Security Conference (2004)
28. Peinado, M., Chen, Y., England, P., Manferdelli, J.: NGSCB: A Trusted Open System. Information Security and Privacy 3108/2004, 86–97 (2004)
29. Pearson, S.: Trusted Computing Platforms: TCPA Technology in Context. Prentice-Hall, Englewood Cliffs (2002)
30. Abadi, M., Wobber, T.: A logical account of NGSCB. In: Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F. (eds.) FORTE 2004. LNCS, vol. 3236, Springer, Heidelberg (2004)
31. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: 13th USENIX Security Symposium (2004)
32. Haldar, V., Chandra, D., Franz, M.: Semantic remote attestation: A virtual machine directed approach to trusted computing. In: USENIX VM (2004)
33. Haldar, V., Franz, M.: Symmetric behavior-based trust: A new paradigm for internet computing. In: New Security Paradigms Workshop (2004)
34. Abadi, M., Blanchet, B.: Secrecy types for asymmetric communication. Theoretical Computer Science 298(3) (2003)
35. Gordon, A.D., Jeffrey, A.S.A.: Authenticity by typing for security protocols. J. Computer Security 11(4) (2003)
36. Fournet, C., Gordon, A., Maffeis, S.: A type discipline for authorization policies. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)
37. Wobber, T., Yumerefendi, A., Abadi, M., Birrell, A., Simon, D.R.: Authorizing applications in Singularity. In: EuroSys 2007: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, Lisbon, Portugal, pp. 355–368. ACM, New York, NY, USA (2007)
38. Wallach, D.S., Appel, A.W., Felten, E.W.: SAFKASI: a security mechanism for language-based systems. ACM Trans. Softw. Eng. Methodol. 9(4) (2000)
39. Abadi, M., Fournet, C.: Access control based on execution history. In: Proceedings of the 10th Annual Network and Distributed System Security Symposium (2003)
40. Cardelli, L.: Program fragments, linking, and modularization. In: POPL 1997 (1997)
41. Bugliesi, M., Focardi, R., Maffei, M.: Compositional analysis of authentication protocols. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, Springer, Heidelberg (2004)
42. Bugliesi, M., Focardi, R., Maffei, M.: Analysis of typed analyses of authentication protocols. In: CSFW (2005)
43. Anderson, R., Kuhn, M.: Tamper resistance - a cautionary note. In: Second USENIX Workshop on Electronic Commerce Proceedings (1996)
44. Huang, A.: Hacking the Xbox. Xenatera Press (2003)
45. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A secure and reliable bootstrap architecture. In: IEEE Symposium on Security and Privacy (1997)
46. Smith, S., Weingart, S.: Building a high-performance, programmable secure coprocessor. Computer Networks, Special Issue on Computer Network Security 31 (1999)
47. Irvine, C., Levin, T.: A cautionary note regarding the data integrity capacity of certain secure systems. In: Integrity, Internal Control and Security in Information Systems (2002)