Dynamic Policy Discovery with Remote Attestation
Extended Abstract

Corin Pitcher[*] and James Riely[**]

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

Abstract. Remote attestation allows programs running on trusted hardware to prove their identity (and that of their environment) to programs on other hosts. Remote attestation can be used to address security concerns if programs agree on the meaning of data in attestations. This paper studies the enforcement of code-identity based access control policies in a hostile distributed environment, using a combination of remote attestation, dynamic types, and typechecking. This ensures that programs agree on the meaning of data and cannot violate the access control policy, even in the presence of opponent processes. The formal setting is a π-calculus with secure channels, process identity, and remote attestation. Our approach allows executables to be typechecked and deployed independently, without the need for secure initial key and policy distribution beyond the trusted hardware itself.

Keywords remote attestation, code-identity based access control, policy establishment, key establishment, π-calculus, Next Generation Secure Computing Base.

1 Introduction

Processes in a distributed system often rely upon the trustworthiness of processes running on other hosts. The remote attestation mechanism in Microsoft's Next Generation Secure Computing Base (NGSCB) [39], in conjunction with trusted hardware specified by the Trusted Computing Group [50, 42], allows processes running on trusted hardware to attach evidence of their identity (and the identity of their environment) to data. Other processes can examine this evidence to assess the degree of trust to place in the process that attested to the data.

Enforcement of access control policies in hostile distributed environments has been a driving concern in the development of trusted hardware and remote attestation. We formalize these notions in a variant of the π-calculus [40], dubbed π-rat, and develop a type system that enforces access control policies in the presence of arbitrary opponents. The type system allows programs to be certified independently and deployed without shared keys or policies beyond those in the trusted hardware.

Organization. In the remainder of this introduction, we set out our goals and assumptions. Section 2 demonstrates the use of π-rat through an extended example. Section 3 presents the dynamics of the language and formally defines runtime errors and

[*] Supported by the DePaul University Research Council.

[**] Supported by the National Science Foundation under Grant No. 0347542.

robust safety. Section 4 develops a type system that ensures safety in the presence of arbitrary attackers, and sketches the proof of robust safety. We conclude with a discussion of related and future work.

Identity and attestation. We assume that processes can be given *identities* in a uniform manner. We write $h[P]$ for a process running with identity h. Initial processes have the form $\#P[P]$, where # is a globally agreed hashing function on process terms. While a process may evolve, its identity cannot. Further, identities cannot be forged. Thus a process Q running with identity $\#P$, must be a residual of P. Our treatment of identities is deliberately abstract; our formal results do not use hash functions. Nonetheless, we write $\#P$ in examples to indicate the identity of a named process. We leave higher-order extensions to our language, internalizing # as an operator, to future work.

In this paper, we do not deal with other forms of identity. For example, there is no notion of code running on behalf of a principal [8], nor is there a notion of explicit distribution [28]. We assume that all resources are globally accessible, for example in a shared heap. Opponents are modeled as processes with access to these globally known resources. As a result, we make no distinction between the many instances of a program; thus $h[P] \mid h[Q]$ is indistinguishable from $h[P \mid Q]$.

At first approximation, an attestation to data M is a signature (with message recovery) upon the pair (h,M), where h is the hash of the process that requested the attestation. The signature is created by trusted hardware using its own private key. The hardware manufacturer issues a certificate for the trusted hardware's public key at the time of manufacture and stores it in the trusted hardware. Upon receipt of an attestation, and the certificate, the relying party verifies the certificate using the manufacturer's well-known public key, and then verifies the signature using the trusted hardware's public key. If successful, the relying party concludes that a process with hash h did request an attestation for M from the trusted hardware. To deduce further properties about M, the relying party must know more about the conditions under which the process with hash h is willing to attest to data.

Policies and certification. We are interested in the distinction between processes that have been certified to obey a certain policy and those that have not been so certified. Realistically, one would like to model multiple kinds of policies and multiple methods of certification; however, here we limit attention to a single, extra-lingual certification, defined as a typing system. We encode policies in types, T, and allow for communication of policy between processes. The particular policies in this paper are access control policies based on code identity. For example, our system allows expression of policies such as "only the ACME media player may display this data".

Opponent processes are those which are not certified. Opponents cannot persuade trusted hardware to create false attestations, but otherwise their behavior is entirely unconstrained. We assume that a conservative approximation of the set of certified processes is available at runtime. That is, a process may inquire, at runtime, whether the process corresponding to a certain identity has been certified. To keep the language simple, we do not deal explicitly with distribution of certifications. In addition, our policies are stated directly in terms of program identities, rather than allowing additional levels of indirection. Both limitations may be alleviated by incorporating a trust management framework, the investigation of which we leave to future work.

Unlike previous analyses of cryptographic protocols [9, 3, 22, 23, 25] remote attestation is intended to be used to establish secure channels starting from insecure channels that are accessible to opponents. In order to allow the communication of policy information during secure-channel establishment, we employ a form of dynamic typing [4, 33]. Our interpretation of $\mathsf{at}(h, M)$ is that h vouches for the policy encoded in M. In particular, if M is $\{N{:}T\}$ with asserted type T, then h vouches that N can safely be used with policy T.

The policy information in attestations from uncertified processes cannot be trusted. While the payload of such an attestation may be stored and communicated, it cannot safely be used in any other way.

Channels and access control. As usual in π, we encode data using channels. Thus access policies regulate the readers and writers of channels. Our policies do not limit possession of channels, only their use; although in the case of an opponent, possession and use are indistinguishable since opponents are not constrained to obey any policy. The policy for a channel is defined when the channel is created and may be communicated over insecure channels via attestations.

In keeping with our high-level interpretation of attestations, we avoid explicit cryptographic primitives [9]. In their place we adopt a polarized variant of the π-calculus [41] which allows transmission of read and write capabilities individually. This simplification is justified by Abadi, Fournet and Gonthier's work on implementing secure channel abstractions [8, 7].

Contributions. In terms of security policies, our aims are modest relative to other recent work on types in process languages. For example, we do not attempt to establish freshness [23, 25] or information flow [30] properties. Nonetheless, we achieve a concise statement of secrecy properties (cf. [1, 3, 17]). For example, if a value is created with type $\mathsf{Data}(h)$, then our typing system ensures that only the program with identity h can display it. Unusually for systems allowing arbitrary opponents [1, 3, 17, 23, 25], our typing system also ensures *memory safety* for certified processes; our approach to opponent typability is reminiscent of [43].

A distinctive aspect of our approach, in keeping with proposed applications of remote attestation, is to minimize reliance upon an authority to distribute keys and policies. For example, in media distribution systems, the executables share nothing but data that is both *public* (can be intercepted by the opponent) and *tainted* (may have come from the opponent). In contrast, spi processes typically require further agreement. Consider trustworthy spi processes P and Q deployed by a mutually-trusted authority that initializes the system "$\mathsf{new}\ kp; (P \mid Q)$." The authority may, for example, create the keypair kp and distribute the public key to P and the private key to Q. A common type environment ensures that P and Q agree on the meaning of data encrypted by kp.

2 Example

A prototypical use of remote attestation is to establish a channel for sending secrets to an instance of a trusted executable, such as a media player that enforces a favored access control policy. A process player on trusted hardware creates a fresh keypair, attests to the public key, then transmits the result to a server. The server verifies the attestation,

Media Player example

```
PWr ≜ Wr ⟨any,#player⟩ (Tnt)
player ≜
 (1) new pch : Ch ⟨any,#player⟩ (Tnt) ;
 (2) wr(sch) ! at(#player, {wr(pch) : PWr}) ;
 (3) rd(pch) ?smsg ;
 (4) let at(s, mdyn) = smsg ;
 (5) iscert s ;
 (6) typecase {m : PData} = mdyn ;
 (7) display m

PData ≜ Data (#player)
server ≜
  (8) repeat rd(sch) ?pmsg ;
  (9) let at(p, chdyn) = pmsg ;
 (10) iscert p ;
 (11) typecase {wr : PWr} = chdyn ;
 (12) new n : PData ;
 (13) wr ! at(#server, {n : PData})
```

concluding that the public key belongs to an instance of the player executable running on trusted hardware. Since the server trusts the player, it encrypts the data, perhaps a movie, and sends the ciphertext back to player. The player is relied upon to enforce a policy, such as not making the data available to other processes, or limiting the number of viewings. The trusted hardware hosting player is relied upon to prevent anyone, including the host's administrator, from violating the player's environment.

We formalize this example at the top of the page. Initially, the player and server agree only on the name of an untrusted (available to the opponent) channel sch, which has type Ch ⟨any, any⟩ (Un) ; the angled brackets contain the channel's policy, the parentheses contain the type of values communicated. The type of sch indicates that anyone (including opponents) may send or receive messages on the channel and that the values communicated are untrusted. The player and server must also have compatible policies for the write capability (representing one key from a keypair) and data, with names PWr and PData respectively. The policies mention the hash of the player program, and thus the two

The player (1) creates a channel of type Ch ⟨any, #player⟩ (Tnt) (representing a keypair), and (2) communicates the write capability (one of the keys) of type PWr to the server by writing on sch. The access control policy associated with the channel pch is ⟨any, #player⟩. The first component any indicates that any executable, certified (typed) or not, may write to the channel; thus the received value is tainted. Using the more restrictive #server as the first component of the policy, meaning that only the server may write to the channel, could be violated after the write capability is communicated on the insecure channel sch. The second component of the policy #player means that only an instance of the player executable can read from the channel.

A more lenient access control policy ⟨any, cert⟩ for pch would allow any well-typed executable, denoted cert, to read from the channel. These two policies illustrate the difference between possession and use in π-rat, because any well-typed executable can possess the read capability for pch—regardless of whether the access control policy is ⟨any, #player⟩ or ⟨any, cert⟩. Both cases are safe because well-typed executables will only use the read capability when they are certain that it is permitted by the access control policy specified by the channel's creator.

The media server (8) repeatedly reads sch. Upon receipt of a message, the server (9) unpacks the attestation in the message, discovering the hash of the attesting process, (10) checks that the hash is certified (the hash of a well-typed executable), then (11) unpacks the payload of the message (the write capability) which involves checking that the stated policy complies with the expected policy. The server then (12) creates a data object and (13) sends it to the player via the write capability. In a similar fashion, (3)-(6) the player receives the data and verifies its origin and policy, then (7) displays the data.

Lines (2) and (13) include attestations. Remote attestation does not allow a remote process to force trusted hardware to identify an uncooperative process. However, processes that are unwilling to identify themselves using attestation may find other processes unwilling to interact with them.

From an implementation perspective, using a hash other than the hash of the enclosing process as the first component of the attestation primitive is unimplementable because trusted hardware will only create attestations with the hash of the requesting process. Due to inherent circularity, it is impossible for an executable to contain its own hash, so we assume that a process is able to query the trusted hardware to find its own hash at runtime: in which case a typechecker implementation would need to verify that the code to perform the query is correct. A more interesting challenge for distributed systems using remote attestation is that two executables cannot contain each other's hashes—one executable may contain the hash of the other executable, as illustrated by the media server code which can only be written after the hash of the player executable is known. Of course, two processes may learn one another's hashes, and incorporate those hashes into policies, during the course of execution.

The media server generates the data with a policy stating that it is only usable by the player. The data stored in n is sent to the player using the write capability wr(pch), so no-one but the player can receive the message. The data is sent inside an attestation, because the player has no reason to trust data that it receives on pch. The type inside the attestation is checked by the player to ensure that it treats the data in accordance with the system's access control policy. When the player receives the hash, it must dynamically check that the hash is that of a well-typed executable. This is necessary to ensure that the type in the attestation is reliable.

The threat considered here is that a process will read or write to a channel in violation of the policy of a well-typed executable that created the channel. For example, we would like to prevent an executable other than player from displaying data n. Our main theorem states that access control violations cannot occur in well-typed configurations, even if the well-typed configuration is placed in parallel with an untyped opponent.

3 Dynamics

We give the syntax and dynamic semantics of π-rat. We describe runtime errors and define safety. We describe types, T, in the next section.

Syntax and Evaluation. The language has syntactic categories for names, terms, processes and configurations. Evaluation is defined in terms of configurations. Assuming a non-colliding hash function (#) on programs — such that if $\#P = \#Q$ then $P = Q$ — initial configurations have the following form.

$$(\#P_1)[P_1] \mid \cdots \mid (\#P_m)[P_m]$$

The configuration represents m concurrent processes, each identified by its hash. This form is not preserved by reduction, since a process may evolve, but its hash does not. (In practice, remote attestation uses the hash of the executable, and the remaining state of the process is ignored.) We thus choose to treat hashes abstractly as names.

Names (*a-z*) serve several purposes. To aid the reader, we use x, y, z to stand for variables, h, g, f to stand for hashes or hash-typed variables, and a, b, c to stand for channels.

Terms and Patterns

$$M,N,L ::= n \mid \mathsf{rd}(M) \mid \mathsf{wr}(M) \mid (M,N) \mid \mathsf{at}(M,N) \mid \{M{:}T\}$$
$$X ::= (x,y) \mid \mathsf{at}(x,y)$$

Terms include names as well as read and write capabilities, $\mathsf{rd}(M)$ and $\mathsf{wr}(M)$, which may be passed individually as in Odersky's polarized π [41]. The term $\mathsf{at}(M,N)$ is an attested message originating from hash M with payload N. The constructors for pairs and attestations each have a corresponding nonblocking destructor in the pattern language. The term $\{M{:}T\}$ carries a term M with asserted type T (cf. the dynamic types of [4]). As illustrated in section 2, terms of the form $\{M{:}T\}$ are used to convey type, and hence policy, information between processes that have no pre-established knowledge of one another's behavior or requirements, but the information can only be trusted when $\{M{:}T\}$ originates from a certified process. Attestation is used to ensure that such terms do originate from a certified process before secure channels are established.

Processes and Configurations

$$P,Q,R ::= \mathsf{iscert}\,M;\,P \mid \mathsf{typecase}\,\{x{:}T\}{=}M;\,P \mid \mathsf{let}\,X{=}M;\,P \mid \mathsf{scope}\,M\,\mathsf{is}\,\sigma$$
$$\mid M?x;\,P \mid M!N \mid \mathsf{new}\,a{:}T;\,P \mid P \,|\, Q \mid \mathsf{repeat}\,P \mid \mathsf{stop}$$
$$C,D,A,B ::= h[P] \mid \mathsf{new}_h\,a{:}T;\,C \mid C \,|\, D \mid \mathsf{stop}$$

The test $\mathsf{iscert}\,M$ succeeds if M is a certified hash, otherwise it blocks. The typecase $\mathsf{typecase}\,\{x{:}T\}{=}M$ succeeds if M is a term with an asserted type that is a subtype of T, otherwise it blocks. The expectation $\mathsf{scope}\,M\,\mathsf{is}\,\sigma$ asserts that the scope of M is limited by the hash formula σ; we discuss hash formulas with runtime errors below. The primitives for reading, writing, new names, concurrency, repetition and inactivity have the standard meanings from the asynchronous π calculus [11, 29]. The constructs for configurations are standard for located π-calculi [28]; note that the construct for new names records the identity of the process that created the name.

Notation. We identify syntax up to renaming of bound names. For any syntactic category with typical element e, we write $fn(e)$ for the set of free names occurring in e. We write $M\{^N\!/\!_x\}$ for the capture-avoiding substitution of N for x in M. For any syntactic category with typical element e, we write sequences as $\vec{e}$ and sets as $\bar{e}$. We occasionally extend this convention across binary constructs, for example writing $\vec{n}{:}\vec{T}$ for the sequence of bindings $n_1 : T_1, \ldots, n_m : T_m$. We sometimes write "_" for a syntactic element that is not of interest. □

The evaluation semantics is given in the chemical style [16] using a structural equivalence and small-step evaluation relation. (We write multistep evaluation as $\bar{g} \triangleright C \rightarrow^{*} D$.) We elide the definition of the structural equivalence, which is standard for located π-calculi [28]; for example, the rule for allowing new to escape from a process is "$h[\mathsf{new}\ a{:}T;\ P] \equiv \mathsf{new}_h\ a{:}T;\ h[P]$".

EVALUATION $(\bar{g} \triangleright C \rightarrow D)$

$$\frac{}{\bar{g} \triangleright f[\mathsf{wr}(a)\,!M] \mid h[\mathsf{rd}(a)?x;\ P] \rightarrow h[P\{^{M}/_{x}\}]} \qquad \frac{}{\bar{g} \triangleright h[\mathsf{iscert}\, f;\ P] \rightarrow h[P]}\ f \in \bar{g}$$

$$\frac{\text{if } h \in \bar{g} \text{ then } \bar{g} : \mathsf{Cert} \vdash T <: S}{\bar{g} \triangleright h[\mathsf{typecase}\ \{x{:}S\} = \{M{:}T\};\ P] \rightarrow h[P\{^{M}/_{x}\}]} \qquad \frac{X\{^{\vec{N}}/_{\vec{y}}\} = M}{\bar{g} \triangleright h[\mathsf{let}\ X = M;\ P] \rightarrow h[P\{^{\vec{N}}/_{\vec{y}}\}]}$$

$$\frac{\bar{g} \triangleright C \rightarrow D}{\bar{g} \triangleright C \mid B \rightarrow D \mid B} \qquad \frac{\bar{g} \triangleright C \rightarrow D}{\bar{g} \triangleright \mathsf{new}_h\ a;\ C \rightarrow \mathsf{new}_h\ a;\ D} \qquad \frac{\bar{g} \triangleright C \rightarrow D}{\bar{g} \triangleright C' \rightarrow D'}\ \begin{array}{l} C \equiv C' \\ D \equiv D' \end{array}$$

The first rule allows communication between processes, in the standard way. The rule for iscert allows a process to verify that a hash is certified; in the residual, f is known to be a certified hash. The rule for typecase allows retrieval of data from a term with an asserted type. A dynamic subtype check enforces agreement between the asserted type T and the expected type S; subtyping is defined in section 4. The let rule is used to decompose attestations and pairs. The structural rules are standard.

Note that $\bar{g}$ is only required by typecase to allow opponents processes to avoid dynamic checks. The other uses of $\bar{g}$ (in iscert and typecase) can be removed, as is shown in the full version of the paper.

Runtime Error and Robust Safety. Our primary interest in typing is to enforce access control policies. Policies are specified in terms of hash formulas.

LATTICE OF HASH FORMULAS $(\rho \leq \sigma)$

$$\rho, \sigma ::= \mathsf{any} \mid \mathsf{cert} \mid h_1, \ldots, h_n \qquad \frac{}{\rho \leq \mathsf{any}} \qquad \frac{}{\bar{h} \leq \mathsf{cert}} \qquad \frac{}{\mathsf{cert} \leq \mathsf{cert}} \qquad \frac{\bar{h} \subseteq \bar{g}}{\bar{h} \leq \bar{g}}$$

Hash formulas are interpreted using an open world assumption; we allow that not all programs nor typed programs are known. The special symbol cert is interpreted as a conservative approximation of the set of well-typed programs.

Access control policies are specified in scope expectations [21, 24]. We develop a notion of *runtime error* to capture access control and memory safety violations.

RUNTIME ERROR $(\bar{g} \triangleright C \xrightarrow{\mathit{error}})$

$$\frac{h \in \bar{g} \quad f \not\leq \sigma\{^{\bar{g}}/_{\mathsf{cert}}\}}{\bar{g} \triangleright h[\mathsf{scope}\, M\ \mathsf{is}\ \sigma] \mid f[M?x;\ P] \xrightarrow{\mathit{error}}} \qquad \frac{h \in \bar{g} \quad M \neq \mathsf{rd}(\_)}{\bar{g} \triangleright h[M?x;\ P] \xrightarrow{\mathit{error}}} \qquad \frac{\bar{g} \triangleright C \xrightarrow{\mathit{error}}}{\bar{g} \triangleright C \mid D \xrightarrow{\mathit{error}}}$$

$$\frac{h \in \bar{g} \quad f \not\leq \sigma\{^{\bar{g}}/_{\mathsf{cert}}\}}{\bar{g} \triangleright h[\mathsf{scope}\, M\ \mathsf{is}\ \sigma] \mid f[M!N] \xrightarrow{\mathit{error}}} \qquad \frac{h \in \bar{g} \quad M \neq \mathsf{wr}(\_)}{\bar{g} \triangleright h[M!N] \xrightarrow{\mathit{error}}} \qquad \frac{\bar{g} \triangleright C \xrightarrow{\mathit{error}}}{\bar{g} \triangleright \mathsf{new}_h\ a;\ C \xrightarrow{\mathit{error}}}$$

$$\frac{h \in \bar{g} \quad M \neq \{\_ : \_\}}{\bar{g} \triangleright h[\mathsf{typecase}\ \{\_ : \_\} = M;\ P] \xrightarrow{\mathit{error}}} \qquad \frac{h \in \bar{g} \quad M \neq (\_, \_)}{\bar{g} \triangleright h[\mathsf{let}\ (\_, \_) = M;\ P] \xrightarrow{\mathit{error}}} \qquad \frac{\bar{g} \triangleright C \xrightarrow{\mathit{error}}}{\bar{g} \triangleright C' \xrightarrow{\mathit{error}}}\ C \equiv C'$$

A runtime error occurs on certain shape errors and whenever a term is used outside of its allowed scope. For example, the following term is in error, since the certified process h is writing on a term of the wrong shape.

$$\{h,g\} \triangleright h[\mathsf{rd}(a)\,!\,n] \xrightarrow{\textit{error}}$$

The following term is also in error, since the certified process h expects the scope of $\mathsf{wr}(a)$ to include only certified processes, yet the uncertified process f is attempting to write on a.

$$\{h,g\} \triangleright h[\mathsf{scope}\ \mathsf{wr}(a)\ \mathsf{is}\ \mathsf{cert}] \mid f[\mathsf{wr}(a)\,!\,n] \xrightarrow{\textit{error}}$$

ROBUST SAFETY

A process is *h-initial* if every attested term has the form "$\mathsf{at}(h, M)$."
A configuration $h_1[P_1] \mid \cdots \mid h_\ell[P_\ell]$ is an *initial $\bar{g}$-attacker* if $\{h_1,\ldots,h_\ell\}$ is disjoint from $\bar{g}$ and every P_i is h_i-initial.
A configuration C is *$\bar{g}$-safe* if $\bar{g} \triangleright C \rightarrow^* D$ implies $\neg(\bar{g} \triangleright D \xrightarrow{\textit{error}})$.
A configuration C is *robustly $\bar{g}$-safe* if $C \mid A$ is $\bar{g}$-safe for every initial $\bar{g}$-attacker A.

The statement of robust safety ensures that certified processes are error-free, even when combined with arbitrary attackers. The restriction that initial attacker $h[P]$ be h-initial requires only that attackers not forge attestations.

4 Statics

We describe a type system that ensures robust safety, i.e., runtime errors cannot occur even in the presence of attackers. Types also convey policy information—access-control policies specified in terms of hash formulas in this paper—that can be transmitted and tested at runtime. *Kinds* are assigned to types to restrict the use of unsafe types. In this section, we present parts of the type system and state the robust-safety theorem. The development is heavily influenced by Gordon and Jeffrey [23] and Haack and Jeffrey [25].

Policies, Types and Kinds. We assign to every channel type a policy regulating access to the channel. For channel types with policy $\langle\rho,\sigma\rangle$, ρ (respectively σ) controls the source (respectively destination) of the data communicated by the channel: thus ρ indicates the set of *writers*; σ indicates the set of *readers*. A *kind* is a policy $\langle\rho,\sigma\rangle$ in which ρ and σ are either cert or any.

POLICIES AND KINDS

$\rho,\ \sigma ::= \mathsf{cert} \mid \mathsf{any} \mid \bar{h}$	Hash Formulas (Repeated)
$\alpha,\ \beta ::= \mathsf{cert} \mid \mathsf{any}$	Kind Formulas
$\Phi, \Psi ::= \langle\rho,\sigma\rangle$	Policies
$\mathcal{K}, \mathcal{J} ::= \langle\alpha,\beta\rangle$	Kinds
$\mathsf{TNT} \triangleq \langle\mathsf{any},\mathsf{cert}\rangle$	Tainted Secret Kind
$\mathsf{PRV} \triangleq \langle\mathsf{cert},\mathsf{cert}\rangle$	Untainted Secret Kind
$\mathsf{UN} \triangleq \langle\mathsf{any},\mathsf{any}\rangle$	Tainted Publishable Kind

$\mathsf{PUB} \triangleq \langle \mathsf{cert}, \mathsf{any} \rangle$	Untainted Publishable Kind
$\dfrac{\rho \leq \rho' \text{ and } \sigma' \leq \sigma}{\langle \rho, \sigma \rangle \leq \langle \rho', \sigma' \rangle}$	Subpolicy Relation

The subpolicy relation, $\Phi \leq \Psi$, indicates that Ψ is more restrictive than Φ. In more restrictive policies, it is always safe to overestimate the origin of a value and to underestimate its scope. That is, in $\langle \rho, \sigma \rangle$, ρ is an upper bound on origin, and σ is an lower bound on scope. When specialized to kinds, the subpolicy relation reduces to the subkinding relation from [25].

$$\begin{tikzcd}
 & \langle \mathsf{any}, \mathsf{cert} \rangle = \mathsf{TNT} & \\
\mathsf{UN} = \langle \mathsf{any}, \mathsf{any} \rangle \arrow[ur] & & \langle \mathsf{cert}, \mathsf{cert} \rangle = \mathsf{PRV} \arrow[ul] \\
 & \langle \mathsf{cert}, \mathsf{any} \rangle = \mathsf{PUB} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

We write $\mathcal{K} \sqcup \mathcal{J}$ for the join operator over this lattice. For example $\mathsf{UN} \sqcup \mathsf{PRV} = \mathsf{TNT}$.

TYPES AND TYPING ENVIRONMENTS

$T, S, U, R ::= \mathsf{Hash} \mid \mathsf{Cert} \mid \mathsf{Top}\,\mathcal{K} \mid \mathsf{Dyn}\,(h)\,\mathcal{K} \mid (x\!:\!T,\, S)$
$\quad \mid \mathsf{Ch}\,\Phi(T) \mid \mathsf{Rd}\,\Phi(T) \mid \mathsf{Wr}\,\Phi(T)$

$\mathsf{Un} \triangleq \mathsf{Top}\,\mathsf{UN}$	Top of Kind UN
$\mathsf{Tnt} \triangleq \mathsf{Top}\,\mathsf{TNT}$	Top of Kind TNT
$E ::= n_1\!:\!T_1, \ldots, n_m\!:\!T_m$	Typing Environments
$dom(\vec{n}\!:\!\vec{T}) \triangleq \vec{n}$	Domain of an Environment

Type Hash can be given to any hash, whereas Cert can be given only to certified hashes. $\mathsf{Top}\,\mathcal{K}$ is the type given to attestations containing data of kind $\mathcal{K}$. $\mathsf{Dyn}\,(h)\,\mathcal{K}$ is the type of dynamically-typed data that was attested by h. In the dependent pair type "$(x\!:\!T,\, S)$" x is bound with scope S; if $x \notin fn(S)$ we write simply $(T,\, S)$. The channel types indicate the policy Φ associated with the channel.

EXAMPLE. Although we allow creation of names at top types, these do not allow a full expression of access control policies. We provide an encoding of data, where $\mathsf{Data}\,(\sigma)$ is the type of data visible to σ.

$$\mathsf{Data}\,(\sigma) \triangleq \mathsf{Wr}\,\langle \sigma, \mathsf{any} \rangle\,(\mathsf{Un}) \qquad \mathsf{display}\, M \triangleq \mathsf{new}\; n\!:\!\mathsf{Un}\,;\; M\,\mathbf{!}\,n$$

A simple use of data is: $\mathsf{new}\; n\!:\!\mathsf{Data}\,(\mathsf{cert})\,;\; \mathsf{display}\; n$. □

Judgments. The following judgments are used in the typing system. Due to space limitations, we discuss only the most important rules.

JUDGMENTS

$E \vdash \diamond$	Well-Formed Typing Environments
$E \vdash \Phi :: \mathcal{K}$	Well-Formed Policies
$E \vdash T :: \mathcal{K}$	Well-Formed Types of Kind $\mathcal{K}$
$E \vdash T <: S$	Subtyping
$E \vdash M : T$	Well-Formed Terms of Type T
$E \vdash_{\overline{h}} P$	Well-Formed Process at h

The rules for environments require that every type in the environment be well-formed. A policy is well-formed with respect to a typing environment if every hash in the policy has type Cert in the environment.

$$\frac{\begin{array}{l} E \vdash \Phi :: \langle \mathsf{cert}, \alpha \rangle \\ E \vdash T :: \langle \_\,, \beta \rangle \quad \alpha \leq \beta \end{array}}{E \vdash \mathsf{Rd}\,\Phi(T) :: \langle \mathsf{cert}, \alpha \rangle} \qquad \frac{\begin{array}{l} E \vdash \Phi :: \langle \mathsf{any}, \alpha \rangle \\ T = \mathsf{Top}\,\langle \mathsf{any}, \beta \rangle \quad \alpha \leq \beta \end{array}}{E \vdash \mathsf{Rd}\,\Phi(T) :: \langle \mathsf{any}, \alpha \rangle}$$

The rules for well-formed types require that read capabilities of kind UN receive values (at a type of) of kind UN; those of kind PUB receive values of kind UN or PUB; those of kind TNT receive values of kind UN or TNT; and those of kind PRV receive values of any kind. Write capabilities are similar for UN and PRV, but differ at the other kinds. Write capabilities of kind PUB send values of kind UN or TNT; those of kind TNT send values of kind UN or PUB. In the analogous rules for write capabilities, the kind is inverted with respect to the policy. As a consequence, if a channel communicates untainted data then the write capability is given at most trusted scope; if a write capability is publishable, then the data it communicates is tainted.

Subtyping. Subtyping is reflexive and transitive, with top types at each kind. Read and write capability types must have related policies in order to be related by subtyping.

$$\frac{\Phi \leq \Psi \quad E \vdash T <: S \quad E \vdash S :: \mathit{kind}(\Psi)}{E \vdash \mathsf{Rd}\,\Phi(T) <: \mathsf{Rd}\,\Psi(S)} \qquad \frac{\Psi \leq \Phi \quad E \vdash S <: T \quad E \vdash S :: \mathit{kind}(\Psi)}{E \vdash \mathsf{Wr}\,\Phi(T) <: \mathsf{Wr}\,\Psi(S)}$$

In the read rule, the requirement that S be well-formed is necessary since Ψ may be tainted even if Φ is not. Likewise in the write rule, Ψ may be publishable even if Φ is not.

Typing and Robust Safety. The typing rules are designed to ensure robust safety whilst allowing typechecked processes to have limited interaction with processes that are not known to be typechecked. The interesting rules for terms are those for dynamic types and attestations.

$$\frac{E \vdash M : T \quad E \vdash h : \mathsf{Cert} \quad E \vdash T :: \mathcal{K}}{E \vdash \{M : T\} : \mathsf{Dyn}(h)\,\mathcal{K}} \qquad \frac{E \vdash M : \mathsf{Cert} \quad E \vdash N : \mathsf{Dyn}(M)\,\mathcal{K}}{E \vdash \mathsf{at}(M, N) : \mathsf{Top}\,\mathcal{K}}$$

The rule for dynamic types constrains each type assertion to be associated with the hash of a typechecked process, and that hash is recorded in the (dependent) dynamic type. The rule for attestations constrains the hash in an attestation to be the same as the one used in the inner type assertion.

Turning to processes, consider the following three rules.

$$\frac{\begin{array}{l} E \vdash M : \mathsf{Top}\,\mathcal{K} \\ E, x{:}\,\mathsf{Hash}, y{:}\,\mathsf{Dyn}\,(x)\,\mathcal{K} \vdash_{\overline{h}} P \end{array}}{E \vdash_{\overline{h}} \mathsf{let\ at}\,(x, y) = M;\ P} \quad \frac{\begin{array}{l} T \in \{\mathsf{Hash}, \mathsf{Cert}\} \\ E, x{:}\,\mathsf{Cert}, E' \vdash_{\overline{h}} P \end{array}}{E, x{:}\,T, E' \vdash_{\overline{h}} \mathsf{iscert}\,x;\ P} \quad \frac{\begin{array}{l} E \vdash M : \mathsf{Dyn}\,(f)\,\mathcal{K} \\ E \vdash f : \mathsf{Cert} \\ E, x{:}\,S \vdash_{\overline{h}} P \end{array}}{E \vdash_{\overline{h}} \mathsf{typecase}\ \{x{:}\,S\} = M;\ P}$$

The pattern-matching rule for attestations is used to decompose an attestation into a hash and a dynamic type associated with that hash. However, the term of dynamic type cannot be unpacked (using the rule for typecase) until the hash is known to correspond to a well-typed process. This is established using iscert, leading to the pattern seen in the example of section 2 of an attestation decomposition, followed by a hash check, and finally a typecase.

$$\frac{E \vdash \diamond \quad E \vdash M : \mathsf{Ch}\,\langle \rho, \sigma \rangle\,(T)}{E \vdash_{\overline{h}} \mathsf{scope\ rd}\,(M)\ \mathsf{is}\ \sigma} \qquad \frac{E \vdash M : \mathsf{Rd}\,\langle \_, \sigma \rangle\,(T) \quad E, x{:}\,T \vdash_{\overline{h}} P \quad h \leq \sigma}{E \vdash_{\overline{h}} M?x;\ P}$$

The type rule for read scoping expectations forces the process making the expectation to know the channel type, and not just the read capability type. This is necessary because policies are invariant on channel types but covariant on read channel types, so a process that only knows the read capability type may have a poor approximation of the actual policy that is used elsewhere in a configuration. To avoid access control violations, the rule for reading processes requires that the process has authorization to read from a read capability. Although a static authorization check may initially appear restrictive, note that the static authorization check may follow a dynamic subtyping check for a read capability received from another process.

The robust safety theorem states that processes can safely be typechecked and deployed independently without any shared untainted or secret data (such as public or secret keys), even in the presence of attackers.

THEOREM (ROBUST SAFETY). *Let E be an environment in which every type is generative and can be given kind* UN. *Let g_i and P_i be defined such that ($1 \leq i \leq n$):*

$$P_i \text{ is } g_i\text{-initial and } E, \overline{h}{:}\,\mathsf{Cert}, \overline{f}{:}\,\mathsf{Hash} \vdash_{g_i} P_i \text{ for some } \overline{h} \subseteq \{g_1, \ldots, g_n\} \text{ and } \overline{f}.$$

Then $g_1\,[P_1] \mid \cdots \mid g_n\,[P_n]$ is robustly $\{g_1, \ldots, g_n\}$-safe.

Proof sketch. The proof requires an invariant that is implied by initial typing and preserved by reduction. We formalize the invariant as a more liberal typing system recording the sets of certified and opponent hashes. The central lemmas are *Certified typability:* All certified processes are well typed. *Opponent typability:* All opponent processes are well typed. *Preservation:* Well typing is preserved by evaluation. *Progress:* Well typed terms cannot give rise to runtime errors. □

5 Related Work

Remote Attestation. NGSCB and the TCG have provoked considerable controversy. For example, see [12, 14].

Abadi [2] 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 data.

The NGSCB [37–39] remote attestation mechanism, and the TCG [50, 42, 15] hardware that underpins it, are more complex than the π-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 [10]. For a concrete account of implementing NGSCB-like remote attestation on top of TCG hardware see [45].

Haldar, Chandra, and Franz [26, 27] 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. In their system, a process requesting an attestation from a second process can send test code to execute on the second process's virtual machine and ask for the results to be reported in attestations. Sadeghi and Stüble [44] 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 [46] consider the use of remote attestation to protect disseminated information.

Process Calculi. As discussed in the introduction, π-rat builds upon existing work [9, 3, 22, 23, 25] 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 [22, 21] should be applicable to π-rat, though we make no attempt to address authenticity or replay attacks here. Finally, our primitive for checking attestation 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.

There is some similarity between our work and that on the distributed π calculus [28]. In Dπ, locations are primarily collections of resources. Here, instead, we view "locations" as principals whose identity is determined by the actual code running. This is a different view of locality, determined less by where the code happens to be running and more on the identity of the code itself.

Because of the close relation between process terms and their hashes, attestation does not appear to fit neatly into existing abstract frameworks for π-calculi, such as applied π [5].

Code Identity. Code identity is also used in stack inspection [51] and other history-based access control policies [6]. Remote attestation can be used to implement similar policies in a distributed environment, but we leave this for future work.

Separate Compilation and Typechecking. The π-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, e.g., [20], 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 [18, 19] have considered separate typechecking in the context of a spi-like calculus.

Trusted Hardware. We have assumed that trusted hardware is trustworthy. Amongst other things, the trusted hardware must correctly protect the memory of processes from attackers, attackers must not be able to access the trusted hardware key, and processor manufacturers must not issue fake certificates and keypairs to anyone (such as law enforcement, intelligence agencies, or data recovery firms). For accounts of the difficulties involved in creating such trusted hardware see [13, 31] for an attacker's perspective and [15, 48] for a defender's perspective. Irvine and Levin [32] provide a warning about placing too much trust in the integrity of COTS.

Other research efforts on implementations of trusted hardware, such as [35, 36, 47], are orthogonal to the work presented here.

6 Conclusion

This paper is an early contribution to the study of remote attestation in programming languages. We have defined an extension of the π-calculus with a remote attestation primitive and access control assertions for channels. Executables may be typechecked and deployed individually, which is a significant advantage for the intended applications of trusted hardware. The resulting typechecked configurations discover and obey access control policies even with the addition of opponents. To the best of our knowledge, this is the first paper to provide static analysis principles for building systems that use the remote attestation mechanism.

By incorporating higher-order communication, one could reason about runtime certification of executables and the distribution of knowledge of the certification. The presence of hashes identifying processes also makes it possible to imagine recovering traditional memory safety without sacrificing opponent typability.

It would be useful to extend π-rat with access control policies using linked namespaces that denote sets of trusted hashes as opposed to sets of public keys in the RT framework [34]. With development tools that also run on trusted hardware, there are some interesting new possibilities. For example, we might use a compiler (not necessarily modeled as a well-typed π-rat executable) that issues an attestation associating the hash of the source code and the hash of the resulting executable. An access control policy might state that a well-typed executable must have been derived from source code signed by a trusted developer's private key, where that developer is expected to follow certain procedures to provide a degree of assurance. The use of development tools that attest to their output would help to mitigate the threat of Trojan horses in tools, see [49].

Acknowledgment. We would like to thank Christian Haack, Radha Jagadeesan, Alan Jeffrey, Will Marrero, William Pollock, Daniel Sweeney, and the anonymous referees for their comments.

References

1. M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5), 1999.
2. M. Abadi. Trusted computing, trusted third parties, and verified communications. In *SEC2004: 19th IFIP International Information Security Conference*, 2004.
3. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3), 2003.
4. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2), 1991.
5. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL '01*, 2001.
6. M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 2003.
7. M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *POPL '00*, 2000.
8. M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Inf. Comput.*, 174(1), 2002.
9. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1), 1999.
10. M. Abadi and T. Wobber. A logical account of NGSCB. In *FORTE '04*, 2004.
11. R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations of the asynchronous π-calculus. *Theor. Comput. Sci.*, 195(2), 1998.
12. R. Anderson. 'Trusted Computing' Frequently Asked Questions. http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html, 2003. Version 1.1.
13. R. Anderson and M. Kuhn. Tamper resistance - a cautionary note. In *Second USENIX Workshop on Electronic Commerce Proceedings*, 1996.
14. W. A. Arbaugh. Improving the TCPA specification. *IEEE Computer*, 2002.
15. W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, 1997.
16. G. Berry and G. Boudol. The chemical abstract machine. In *POPL '90*, 1990.
17. M. Bugliesi, S. Crafa, A. Prelic, and V. Sassone. Secrecy in untrusted networks. In *ICALP '03*, 2003.
18. M. Bugliesi, R. Focardi, and M. Maffei. Compositional analysis of authentication protocols. In *ESOP*, 2004.
19. M. Bugliesi, R. Focardi, and M. Maffei. Analysis of typed analyses of authentication protocols. In *CSFW*, 2005.
20. L. Cardelli. Program fragments, linking, and modularization. In *POPL '97*, 1997.
21. C. Fournet, A. Gordon, and S. Maffeis. A type discipline for authorization policies. In *ESOP '05*, 2005.
22. A. D. Gordon and A. S. A. Jeffrey. Authenticity by typing for security protocols. *J. Computer Security*, 11(4), 2003.
23. A. D. Gordon and A. S. A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *J. Computer Security*, 12(3/4), 2004.
24. A. D. Gordon and A. S. A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *CONCUR*, 2005.
25. C. Haack and A. S. A. Jeffrey. Pattern-matching spi-calculus. In *Proc. IFIP WG 1.7 Workshop on Formal Aspects in Security and Trust*, 2004.
26. V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *USENIX VM*, 2004.

27. V. Haldar and M. Franz. Symmetric behavior-based trust: A new paradigm for internet computing. In *New Security Paradigms Workshop*, 2004.
28. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173, 2002.
29. K. Honda and M. Tokoro. On asynchronous communication semantics. In *ECOOP '91: Proceedings of the Workshop on Object-Based Concurrent Computing*, 1992.
30. K. Honda, V. T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *ESOP '00*, 2000.
31. A. Huang. *Hacking the Xbox*. Xenatera Press, 2003.
32. C. Irvine and T. Levin. A cautionary note regarding the data integrity capacity of certain secure systems. In *Integrity, Internal Control and Security in Information Systems*, 2002.
33. X. Leroy and M. Mauny. Dynamics in ML. In *FPCA*, 1991.
34. N. Li and J. Mitchell. RT: A role-based trust-management framework. In *DARPA Information Survivability Conference and Exposition (DISCEX III)*, 2003.
35. D. Lie, C. Thekkath, P. Lincoln, M. Mitchell, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS-IX*, 2000.
36. D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *19th ACM Symposium on Operating Systems Principles*, 2003.
37. Microsoft. Longhorn developer preview documentation. Distributed at Microsoft's Professional Developers Conference in Los Angeles, 2003.
38. Microsoft. NGSCB: TCB and software authentication, 2003.
39. Microsoft. Security model for the Next-Generation Secure Computing Base, 2003.
40. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1), 1992.
41. M. Odersky. Polarized name passing. In *FST-TCS '95*, 1995.
42. S. Pearson, editor. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall, 2002.
43. J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. *J. Automated Reasoning*, 31(3–4), 2003.
44. A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *New Security Paradigms Workshop*, 2004.
45. R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *13th USENIX Security Symposium*, 2004.
46. R. Sandhu and X. Zhang. Peer-to-peer access control architecture using trusted computing technology. In *SACMAT*, 2005.
47. A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.
48. S. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31, 1999. Special Issue on Computer Network Security.
49. K. Thompson. Reflections on trusting trust. *CACM*, 27(8), 1984.
50. Trusted Computing Group. Trusted Computing Platform Alliance: Main specification, version 1.1b. http://www.trustedcomputinggroup.org, 2003.
51. D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol.*, 9(4), 2000.