Reflections on Trust
Trust Assurance by Dynamic Discovery of Static Properties

Andrew Cirillo and James Riely[⋆]

DePaul University, School of Computing
{acirillo,jriely}@cs.depaul.edu

Abstract. Static analyses allow dangerous code to be rejected before it runs. The distinct security concerns of code providers and end users necessitate that analysis be performed, or at least confirmed, during deployment rather than development; examples of this approach include bytecode verification and proof-carrying code. The situation is more complex in multi-party distributed systems, in which the multiple web services deploying code may have their own competing interests. Applying static analysis techniques to such systems requires the ability to identify the codebase running at a remote location and to dynamically determine the static properties of a codebase associated with an identity. In this paper, we provide formal foundations for these requirements. Rather than craft special-purpose combinators to address these specific concerns, we define a reflective, higher-order applied pi calculus and apply it. We treat process abstractions as serialized program files, and thus permit the direct observation of process syntax. This leads to a semantics quite different from that of higher-order pi or applied pi.

1 Security in Distributed Open Systems

In an *open system*, program code is under the control of mutually distrusting parties prior to deployment. Local software security may be maintained in such a system by using dynamic verification at load time, rejecting code that fails analysis. For example, a client browser may validate embedded scripts before execution; a server application may validate SQL queries derived from client input. It is common for virtual machines to perform bytecode verification on class files loaded from remote sources [1]. Similar approaches are taken in [2, 3].

Such analysis can establish *local* properties, for example, that unauthorized code does not gain access to sensitive system resources. It is more difficult to obtain *global* security guarantees, since no single observer has access to all of the required code. Consider the simplest possible such system, consisting of a client and server. The client may wish to ensure that sensitive data does not escape the server. Note that the client's trust in the organization running the server is not sufficient—the client must also trust the software running on the server. If the software is buggy, the client may need to trust all other clients as well. The client may not require a full proof of correctness on the part of the server, but may be satisfied to know that the server's runtime system has all current security patches applied or that it performs some simple integrity checks on data

[⋆] This work was supported by the National Science Foundation under Grant No. 0347542.

supplied by other users. The server has symmetric concerns, for example, restricting client software in order to establish non-repudiation of a commercial transaction.

In current practice, attempts to establish such global properties are ad hoc and informal: “If I only give my credit card number to pay-pal, everything will be fine.” The biggest flaw of such policies is not that they lack formality, but that they are overly restrictive. Lesser known vendors are high risk simply because they are lesser known.

Trusted computing [4] has the potential to enable less restrictive policies. Systems that use trusted computing and *remote attestation* [5] conditionalize their interactions with currently running, but physically distant, processes based on the identity of the program code the remote party is running. Secure messages identify the code of their senders; recipients trust the contents based on static properties of the senders' code.

In prior work [6], we made a first step toward formalizing such systems, developing a higher-order π calculus with ad hoc primitives for remote attestion and a type system that enforced memory safety in the face of arbitrary attackers. Here we improve this work by generalizing both the primitives of the language and the policies to which it applies; we also provide a more powerful and realistic attacker model.

In practice, there are several operations available on an executable: (a) one can communicate it as data, (b) one can execute it, (c) one can identify it by comparing it syntactically to another value, (d) one can extract data from it, or (e) one can disassemble it and operate on its components. Operations (a) and (b) are features of what is commonly refered to as higher order programming. Operations (c) through (e), which expose the syntax of mobile code, are features of what might be called introspective, or reflective programming.

Formalizing many aspects of open systems requires a reflective approach; trusted computing requires at least syntactic identification of code, dynamic verification also requires disassembly. While identification and data extraction are reasonably straightforward operations (see e.g., [6]), modeling the disassembly of an executable can be complicated. For example, if primitive destructors for process syntax are used one must take special precautions to keep names and variables from escaping their scopes, and also to ensure that syntax is preserved by substitution.

Our key observation is that all three can be represented by extending higher order π with pattern matching on abstractions. Our interest is to interalize static analysis at the level of specification, rather than implementation. We are thus able to restrict pattern variables to match subvalues, rather than subvalues *and* subprocesses. The langauge can encode arbitrary calculations on the syntax of an abstraction by “guessing” the structure of the program and substituting pattern variables for values that are not known a priori. To make up for the loss of induction over process syntax, we allow an infinite number of such processes in parallel. The language can thus model abstract specifications of verifiers for static properties.

A language which allows explicit decomposition of processes has recently been proposed by Sato and Sumii [7]; the language considered here represents a middle-ground, giving a simpler syntax and semantics but with a slight cost in terms of expressiveness. In particular, while we can model arbitrary verifiers, we do not permit the verifiers themselves to be treated as programs, which would then be subject to verification.

The paper is structured as follows. In Section 2, we present the syntax and operational semantics of the language. In Section 2, we then develop a systematic method for describing processes that perform dynamic verification. In Section 3 we apply the theory to a simple type system that guarantees memory safety. In Section 4 we apply it to a trusted computing platform.

2 A Reflective Pattern-Matching π-Calculus

In the face of attackers, the higher-order π calculus (HOπ) [8–10] raises subtle issues. Consider the following example, where *pub* represents a public channel, accessible to all parties including attackers, and *passwd* a channel accessible to only Alice and Bob.

$$\begin{aligned}
\mathsf{Alice} &\triangleq \nu secret.pub!((x,y)\mathsf{if}\ x = passwd\ \mathsf{then}\ y!secret)\\
\mathsf{Bob} &\triangleq pub?(prog)\nu b.(prog \cdot (passwd, b) \mid b?(z)Q)\\
\mathsf{Sys} &\triangleq \nu pub.(\mathsf{Mallory} \mid \nu passwd.(\mathsf{Alice} \mid \mathsf{Bob}))
\end{aligned}$$

Alice creates a secret name (*secret*) and embeds it in an abstraction, which is then written on the public channel. If the first argument x of the abstraction matches *passwd* then the secret is written on the second argument y. Bob reads the code from the public channel and instantiates it with *passwd* and a callback channel. After unlocking the secret Bob continues as Q with z bound to *secret*.

Consider an arbitrary attacker, Mallory, who knows *pub* but not *passwd*. Because Mallory has access to *pub*, he can intercept Alice's program before it is received by Bob. Once in possession of the program, Mallory need only inspect its contents to extract the embedded secret without executing the code, thus circumventing the password check.

HOπ does not model this sort of attack, since HOπ abstractions may only be communicated as data or run. By analogy to object [11] and class [12, Ch. 5] serialization, HOπ allows process abstractions to be serialized, but does not allow for inspection of the serialized form. Nonetheless, such attacks are of direct relevance to practical systems [13], therefore in this section we extend HOπ with reflection features.

Reflective π. We define a local, value-passing, asynchronous higher-order π parameterized over a signature that specifies value constructors. A general-purpose pattern-matching destructor works for any kind of value, including abstractions. As in pattern matching spi [14], we equip pattern matching with a notion of *Dolev-Yao derivability* that gives a semantics to cryptographic primitives by restricting patterns to those that represent implementable operations. The resulting language is simple, yet powerful.

Syntax and Operational Semantics. A *value signature* (Σ) comprises three components: a set of value constructors (f), a sorting that assigns each constructor an arity, and a Dolev-Yao derivability judgement ($\Vdash$) that constrains value patterns. Fix a value signature. Values include names, variables, process abstractions and constructor applications; processes include stop, local input, asyncronous output, application, parallel composition, restriction, replication, value construction and a pattern matching destructor. We also allow some processes to contain infinite parallel components.

REFLECTIVE π

Syntax:

$$L,M,N,S,T \ ::= a \mid x \mid (x)P \mid f(\widetilde{M})$$

$$O,P,Q,R \ ::= 0 \mid a?N \mid M!N \mid M \cdot N \mid \Pi_i P_i \mid \nu a.P \mid *P \mid \mathsf{let}\, x = f\langle\widetilde{M}\rangle \,\mathsf{in}\, P$$
$$\mid \mathsf{case}\, M \,\mathsf{of}\, \exists\widetilde{x}.N \,\mathsf{in}\, P \quad \textit{where } fn(N), (fv(N) - \widetilde{x}), N \Vdash \widetilde{x}$$

Reduction Axioms:

$$\text{(COMM)} \quad a?M \mid a!N \longrightarrow M \cdot N$$

$$\text{(APP)} \quad ((x)P) \cdot N \longrightarrow P\{x := N\}$$

$$\text{(CONST)} \quad \mathsf{let}\, x = f\langle\widetilde{M}\rangle \,\mathsf{in}\, P \longrightarrow P\{x := f(\widetilde{M})\}$$

$$\text{(CASE)} \quad \mathsf{case}\, M\{\widetilde{x} := \widetilde{N}\} \,\mathsf{of}\, \exists\widetilde{x}.M \,\mathsf{in}\, P \longrightarrow P\{\widetilde{x} := \widetilde{N}\}$$

We distinguish variables from names, allowing input only on names; therefore only output capabilities may be communicated. This restriction makes examples simpler but is not essential to the theory. We require that process abstractions have finite syntax except when they are used as the right-hand side of an input process. The name a is bound in "$\nu a.P$" with scope P. The variable x is bound in "$(x)P$" and in "$\mathsf{let}\, x = f\langle\widetilde{M}\rangle \,\mathsf{in}\, P$" with scope P. The variables $\widetilde{x}$ are bound in "$\mathsf{case}\, M \,\mathsf{of}\, \exists\widetilde{x}.N \,\mathsf{in}\, P$" with scope N and P. Let fn and fv return free names and variables, respectively. Identify syntax up to renaming of bound names and variables. Write "$P\{x := M\}$" and "$N\{a := M\}$" for the capture-avoiding substitution of M for x in P and M for a in N. A constructor application, $f(\widetilde{M})$, is *well-sorted* if $|\widetilde{M}|$ matches the arity of f. Constructor applications in both the value and process languages are assumed to be well sorted, as in applied π [15].

The variables $\widetilde{x}$ are *pattern bound* in $\exists\widetilde{x}.N$ with scope N. We say that $\exists\widetilde{x}.N$ is a *well-formed pattern* if $\widetilde{x} \subseteq fn(N)$. A term (or process) is *well-formed* if every pattern it contains is well-formed and if any variable x that occurs under a constructor application is pattern bound by an enclosing pattern. For example, "$\mathsf{case}\, M \,\mathsf{of}\, \exists x.f(x) \,\mathsf{in}\, 0$" is well-formed, but "$\mathsf{case}\, M \,\mathsf{of}\, \exists x.f(x) \,\mathsf{in}\, a!f(x)$" and "$(x)a!f(x)$" are not well-formed. In the sequel, we assume that all terms are well-formed.

Note that while first order value passing languages, such as applied π [15], are often abstract with respect to the time at which a value is constructed, mixing reflection and cryptography requires that we distinguish the code that creates a value from the value itself. As an example, suppose "$\mathsf{enc}(M,N)$" represents encryption of M with key N and consider the abstraction "$(x)a!\mathsf{enc}(x,b)$"; the missing payload implies that the encryption has not yet taken place, in which case an observer should be able to extract b. Similarly in "$(x)a!\mathsf{enc}(b,x)$" we expect b to be visible. The case of "$(x)a!\mathsf{enc}(b,b')$" is, however, ambiguous; if it represents a program that does an encryption then both b and b' should be visible, but if it represents a program embedded with an already-encrypted message then neither should be visible. We resolve this ambiguity by providing an explicit construction call in the process language and requiring that constructor applications in the value language contain no free (non-pattern) variables.

The pattern-matching destructor "$\mathsf{case}\, M \,\mathsf{of}\, \exists\widetilde{x}.N \,\mathsf{in}\, P$" allows nested matching into constructed values and abstractions. We require that all bound pattern variables ($\widetilde{x}$) occur at least once in N, and they may occur more than once. To match, all occurences of a pattern variable must match identical values. When matching abstractions we assume that pattern variables are always chosen so as not to conflict with variables bound by the abstraction.

Patterns are also constrained by the Dolev-Yao derivability judgement. The judgement "$\widetilde{M} \Vdash \widetilde{N}$" expresses that the values $\widetilde{N}$ can be constructed by agents with knowledge of the values $\widetilde{M}$. We then require that pattern variables be derivable from the terms mentioned explicitly in the pattern. For example, a sensible derivability judgement might include "$\mathsf{enc}(x,M) \Vdash x$," which would allow decryption when the key is specified, but not "$\mathsf{enc}(x,y) \Vdash x,y$," which would allow extracting both the contents and the key of an encrypted message without specifying the key.

For clarity, we make use of a more concise syntax in written examples by observing the following notational conventions. We omit binders from patterns clauses when they are clear from context (as in $\mathsf{case}\ M\ \mathsf{of}\ (x,y)\ \mathsf{in}\ P$). We omit unused bound variables, writing $()P$ for $(x)P$ when $x \notin fn(P)$. We omit explicit let binders when the meaning is clear, for example writing "$a!f\langle x\rangle$" for "$\mathsf{let}\ y = f\langle x\rangle\ \mathsf{in}\ a!y$." We also assume that a value constructor for pairs is available and use the obvious derived forms for tuples.

As usual, operational semantics are described in terms of separate structural equivalence and reduction relations. We elide the definition of structural equivalence and the context rules for reduction, which are entirely standard for π calculi, and present only the reduction axioms. COMM brings an abstraction and an argument together over a named channel; APP applies an argument to an abstraction, substituting the argument for the parameter variable; and CONST constructs a new value from a constructor symbol and a series of arguments. CASE allows a pattern match to proceed only if the value is syntactically identical (up to α-equivalence) to the pattern modulo a substitution for the bound variables of the pattern. For example, the pattern $\exists x.(y)a!x$ does not match $(y)a!(y,b)$ because the substitution of (y,b) for x would capture y, however the pattern $\exists x.(z)a!(z,x)$ does match because the bound z can be renamed to y.

Equivalences. Behavioral equivalences are not the focus of this paper (see [10] for a thorough introduction), however we very briefly note that adding reflection to HOπ in almost any capacity will have a dramatic effect on its equivalences. In particular, any equivalence closed under arbitrary contexts, which may have holes under abstraction binders, collapses immediately to syntactic identity.

An interesting equivalence would therefore only consider contexts without holes in abstractions (these could be called *non-value* contexts). Since they are transparent, passing process abstractions in this context is no different than for any ordinary values such as pairs or integers, hence the standard definitions for value-passing π-calculi [10, Sec 6.2] can be used. While complications do arise in the presence of non-transparent (i.e., cryptographic) values, these issues are orthogonal to higher-orderness and reflection and have already been addressed in the literature [16, 15].

Embedded Password Attack Revisited. We now reconsider the example above and see that, as desired, it is not secure. Consider an attacker, Mallory, defined as follows.

$$\mathsf{Mallory} \triangleq pub?(prog)\mathsf{case}\ prog\ \mathsf{of}\ \exists(z_1,z_2).((x,y)\mathsf{if}\ x = z_1\ \mathsf{then}\ y!z_2)\ \mathsf{in}\ (\ldots)$$

As was the case with only a higher-order features, Mallory is able to intercept the program file with an input on the public channel, *pub*. By using reflection, however, Mallory is now also able to extract both the password and secret without running the program. The continuation $(\ldots)$ has z_1 bound to *password* and z_2 bound to *secret*.

Dynamic Verification. A *verifiable property* is a property of abstractions that can be decided by a static analysis tool that is invoked at runtime. We call such tools *verifiers*. The proper use of a verifier can, for example, ensure the safety of a process even when it executes code obtained from an untrusted source.

Formally, a verifiable property is a predicate on finite abstractions subject to the following constraints. First, it must be at least semi-decidable. Second, we require that it depend on a specific usage of only a finite set of names. Given a property, $\mathcal{P}$, and a set of names, S, we say that *S supports $\mathcal{P}$* if for every $M \in \mathcal{P}$ and every $a, b \notin S$ where $a \notin fn(M)$, $M\{b := a\} \in \mathcal{P}$. We write $supp(\mathcal{P})$ for the smallest set of names that supports $\mathcal{P}$ and restrict our attention to properties that have $supp(\mathcal{P})$ finite. As an example of a predicate on values without finite support, impose an total ordering on infinite subset of names n_i such that $n_i < n_{i+1}$ and consider the predicate that insists that only n_{i+1} may be output on n_i. Such a predicate is not interesting to us, since names have no inductive structure and therefore one cannot define an algorithm to decide it.

It is relatively easy to describe processes that implement verifiers using infinite syntax. Treating properties as sets of values, we quantify clauses over elements of the set. Note, however that it is not quite as simple as specifying one pattern that exactly matches each element of the set. For example, the naive verifier,

$$a?((z))\Pi_{M\in\mathcal{P}}\big(\mathsf{case}\ z\ \mathsf{of}\ M\ \mathsf{in}\ ((x)P\cdot z)\big) \mid \Pi_{N\notin\mathcal{P}}\big(\mathsf{case}\ z\ \mathsf{of}\ N\ \mathsf{in}\ Q\big)$$

inputs a value to be checked and then pattern matches all values that satisfy the property continuing as P with the value bound to x, and all values that do not satisfy the property continuing as Q. Quantification over all values, however, means that such a process would reference not just an infinite subset of names but the whole universe of names, thus violating important assumptions about bound names. With a little more effort, though, we can build a verifier that has a finite set of free names provided that the underlying property has finite name support, as in the following derived form.

DERIVED FORM: VERIFY

$$\begin{aligned}\mathsf{verify}\ M\ \mathsf{as}\ \mathcal{P}(x)\ \mathsf{in}\ P\ \mathsf{else}\ Q \triangleq \nu b.\ \big(\ &\Pi_{N\in\mathcal{P}}(\mathsf{case}\ M\ \mathsf{of}\ \exists\widetilde{z}.N\{\widetilde{a} := \widetilde{z}\}\ \mathsf{in}\ b?()((x)P\cdot N\{\widetilde{a} := \widetilde{z}\}))\\ \mid\ &\Pi_{L\notin\mathcal{P}}(\mathsf{case}\ M\ \mathsf{of}\ \exists\widetilde{z}.L\{\widetilde{a} := \widetilde{z}\}\ \mathsf{in}\ b?()Q) \mid b!b\ \big)\\ \textit{where}\ \widetilde{a} = fn(M) - supp(\mathcal{P})\ ,\ &\widetilde{z} \cap (fv(P) \cup fv(Q) \cup \{x\}) = \emptyset\ \textit{and}\ |\widetilde{z}| = |\widetilde{a}|\end{aligned}$$

Now $fn(\mathsf{verify}\ M\ \mathsf{as}\ \mathcal{P}(x)\ \mathsf{in}\ P\ \mathsf{else}\ Q) = supp(\mathcal{P}) \cup fn(M) \cup fn(P) fn(Q)$, hence it is finite if and only if $supp(\mathcal{P})$ is finite and M, P, Q have finite free names. The elimination of uninformative names from patterns allows finite name dependency, but also causes some of the pattern clauses under the quantification to overlap. We can be assured that this overlap is safe because names outside of $supp(\mathcal{P})$ by definition cannot affect the satisfaction of the property, hence true patterns may only overlap with other true patterns and false with false. The use of b as a signal channel prevents more than one clause from executing so the behavior resembles that of a naive implementation.

Note that this representation is general enough to allow one to express verifiers for a wide range of analyses and properties, including even those that may not be finitely realizable. In particular, when properties are only semi-decidable this representation will be unrealistically powerful, however for the purpose of establishing safety theorems the approach is adequate.

3 Typability as a Verifiable Property

The framework for dynamic verification presented above may be applied to any verifiable property. A verifiable property is not necessarily a useful security property. For example, it is verifiable that an executable is signed, but this does not impart any security in itself. To establish a security theorem of some sort, we must choose a property with provable security guarantees.

In this section we consider an example of such a property, formalizing a common approach to typing that guarantees the absence of certain runtime errors in the presence of arbitrary attackers [17–21]. Typability in this type system represents a verifiable property subject to implementation as an analysis procedure that can be invoked at runtime. To provide support for interesting examples, we use a signature that includes some basic constructs that are useful in open distributed systems, including dynamically typed messages [22] and cryptographic hashes and symmetric-key encryption.

The novelty is not in the type system itself, which is mostly standard, so much as how it serves as an example for dynamic verification. For this reason we simplify the typed language by supporting nested pattern matching only when extracted values can be treated at type Top, and type-safe pattern matching only for top-level patterns. Many of these restrictions can be eased using, for example, techniques developed in [14].

SIGNATURE (Σ)

Value Constructors (where f^k is a constructor f of arity k):

$$\Sigma = \mathsf{unit}^0,\ \mathsf{pair}^2,\ \mathsf{dyn}^2,\ \#^1,\ \mathsf{enc}^2,\ \rightarrow^2,\ \mathsf{Unit}^0,\ \times^2,\ \mathsf{Dyn}^2,\ \mathsf{Hash}^1,\ \mathsf{Un}^0,\ \mathsf{Top}^0,\ \mathsf{Ch}^1,\ \mathsf{Key}^1$$

Derivability Rules:

$$\frac{}{\widetilde{M}, N \Vdash N} \qquad \frac{\widetilde{M} \Vdash N_1 \ \ldots\ \widetilde{M} \Vdash N_k}{\widetilde{M} \Vdash N_1, \ldots, N_k} \qquad \frac{\widetilde{M} \Vdash \widetilde{N}}{\widetilde{M} \Vdash f(\widetilde{N})} \qquad \frac{\widetilde{M}, \widetilde{N} \Vdash \widetilde{L} \quad f \notin \{\#, \mathsf{enc}\}}{\widetilde{M}, f(\widetilde{N}) \Vdash \widetilde{L}} \qquad \frac{\widetilde{M} \Vdash N', \widetilde{L}}{\widetilde{M}, \mathsf{enc}(N, N') \Vdash N, \widetilde{L}}$$

Language. Assume a signature with the following values: unit and pair, which work as usual; $\mathsf{dyn}(M,T)$, for dynamically-typed message that asserts that M is a value of type T; $\#(M)$ for the cryptographic hash of M; $\mathsf{enc}(M,N)$ for the message M encrypted with key N; and type constructors Unit, Un, Top, $\mathsf{Ch}(T)$, $T \rightarrow \mathsf{Proc}$, $\mathsf{Hash}(T)$, $\mathsf{Key}(T)$, and $\mathsf{Dyn}(M,T)$. We write "(M,N)" as shorthand for "$\mathsf{pair}(M,N)$," we write abstraction types postfix, as in "$T \rightarrow \mathsf{Proc}$," and we write "Dyn" for "$\mathsf{Dyn}(\mathsf{unit}, \mathsf{Unit})$."

Derivability rules exclude only patterns that would allow one to derive the original value of a cryptographic hash or the contents of an encrypted message without the key. We elide the derivability rules for processes since process syntax is always transparant.

Since they appear in dynamically typed messages, types are nominally first-class values. Informally, we use T,S for values that represent types, however note that there is no dedicated syntactic category for type values. Our treatment of dynamic typing is standard except for our use of the type $\mathsf{Dyn}(M,T)$, which is explained later.

We avoid annotating processes with types primarily so we do not have to commit to whether annotations should be visible to inspection or not (in comparison to untyped machine code vs. typed bytecode). Annotations can instead be coded up using dynamically typed messages. We write "$\nu(a:T)P$" for "$\nu a.\mathsf{let}\ x = \mathsf{dyn}\langle a,T\rangle\ \mathsf{in}\ P$" where $x \notin fn(P)$ when we wish to force the typechecker to commit to a specific type or simply add clarity.

Safety and Robust Safety. Our objective is simply to prevent the misuse of a fixed set of typed initial channels. Let the metavariable $\mathcal{T}$ range over a language of types that includes type values, plus the non-first class type TYPE. A type environment (Γ) binds names and variables to types in the usual fashion; we write "$\Gamma \ni a : \mathcal{T}$" to mean that $\Gamma = \Gamma', a : \mathcal{T}, \Gamma''$ and $a \notin dom(\Gamma'')$. An *initial typing* is a type environment taking a set of initial channels to channel types. An error occurs if a process violates the contract of an initial channel by writing a non-abstraction value on a channel with a type of the form $\mathsf{Ch}(T \to \mathsf{Proc})$. Our focus on shape errors involving abstractions is arbitrary; other errors are also possible.

Let Δ be an initial typing with domain $a_1,\ldots,a_n$. We say that a process P is Δ-*safe* if whenever $P \Longrightarrow \nu\widetilde{b}.(a_i?M \,|\, a_i!N \,|\, Q)$ and $\Delta(a_i) = \mathsf{Ch}(T \to \mathsf{Proc})$, N is of the form $(x)R$. We say that a process O is an *initial* Δ-*opponent* if for all $a \in (fn(O) \cap dom(\Delta))$, $\Delta(a) = \mathsf{Ch}(\mathsf{Un})$. We say that P is *robustly* Δ-safe if $(O \,|\, P)$ is safe for an arbitrary initial Δ-opponent O.

Type System. We now present a type system that enforces robust safety. The system includes type judgements for well-formed values and well-formed processes.

The rules for well-formed values are mostly standard: hashes of values of type T type at $\mathsf{Hash}(T)$; names that are used as signing keys for values of type T type at $\mathsf{Key}(T)$; encrypted messages type at Un and require that the content type be compatible with the key type. The one novelty is in the rules for dynamically typed messages, which allow a forwarder to delegate part of the task of judging the trustworthiness of a message to the recipient. A message $\mathsf{dyn}(M,T)$ types at $\mathsf{Dyn}(N,S)$ if either M can be typed at T, or N cannot be typed at S. Opponent values are constructed from names that type at $\mathsf{Ch}(\mathsf{Un})$, cryptographic hashes and encrypted messages.

The rules for well-formed processes are similarly standard, except for the rules for pattern matching. Specific rules are defined for top-level (non-nested) pair splitting, typecase and decryption operations. A separate general-purpose rule permits pattern matching with arbitrarily nested patterns but restricts pattern variables to Top.

The type rules support the use of dynamic types to authenticate data based on the trust placed in the program that created it. For example, the type $\mathsf{Dyn}(\#(N), \mathsf{Hash}(S \to \mathsf{Proc}))$ can be given to messages that are known to have been received from a residual of the abstraction N applied to an argument of type S. If the identity but not typability of the sender is known, a forwarder can thus record the (code) identity of the sender without

WELL-FORMED VALUES ($\Gamma \vdash M : \mathcal{T}$)

Trusted Values:

$$\frac{\Gamma \ni a,x : \mathcal{T}}{\Gamma \vdash a,x : \mathcal{T}} \qquad \frac{\Gamma, x : \mathcal{T} \vdash P}{\Gamma \vdash (x)P : \mathcal{T} \to \mathsf{Proc}} \qquad \Gamma \vdash \mathsf{unit} : \mathsf{Unit} \qquad \frac{\Gamma \vdash M : \mathcal{T} \quad \Gamma \vdash N : \mathcal{S}}{\Gamma \vdash (M,N) : \mathcal{T} \times \mathcal{S}}$$

$$\frac{\Gamma \vdash T : \textsc{Type}}{\Gamma \vdash \mathsf{Ch}(T) : \textsc{Type}} \qquad \frac{\Gamma \vdash T : \textsc{Type}}{\Gamma \vdash T \to \mathsf{Proc} : \textsc{Type}} \qquad \Gamma \vdash \mathsf{Top}, \mathsf{Un}, \mathsf{Unit}, \mathsf{Dyn} : \textsc{Type}$$

$$\frac{\Gamma \vdash T : \textsc{Type}}{\Gamma \vdash \mathsf{Hash}(T) : \textsc{Type}} \qquad \frac{\Gamma \vdash T : \textsc{Type}}{\Gamma \vdash \mathsf{Key}(T) : \textsc{Type}} \qquad \frac{\Gamma \vdash M : \mathcal{T} \quad \Gamma \vdash S : \textsc{Type}}{\Gamma \vdash \mathsf{Dyn}(M,S) : \textsc{Type}}$$

$$\frac{\Gamma \vdash M : \mathcal{T}}{\Gamma \vdash M : \mathsf{Top}} \qquad \frac{\Gamma \vdash T : \textsc{Type} \quad \Gamma \vdash M : T \quad \Gamma \vdash N : \mathcal{S}}{\Gamma \vdash \mathsf{dyn}(M,T) : \mathsf{Dyn}(N,\mathcal{S})} \qquad \frac{\Gamma \vdash M : \mathcal{T}}{\Gamma \vdash \#(M) : \mathsf{Hash}(\mathcal{T})}$$

$$\frac{\Gamma \vdash M : \mathcal{T} \quad \Gamma \vdash N : \mathsf{Key}(\mathcal{T})}{\Gamma \vdash \mathsf{enc}(M,N) : \mathsf{Un}}$$

Opponent Values:

$$\frac{\Gamma \vdash a : \mathsf{Ch}(\mathcal{T}) \quad \mathcal{T} \in \{\mathsf{Un}, \mathsf{Top}\}}{\Gamma \vdash a : \mathsf{Un}} \qquad \frac{\Gamma, x : \mathsf{Un} \vdash P \quad (\forall a \in fn(P))\, \Gamma \vdash a : \mathsf{Un}}{\Gamma \vdash (x)P : \mathsf{Un}}$$

$$\frac{\Gamma \vdash \widetilde{M} : \mathsf{Un}}{\Gamma \vdash f(\widetilde{M}) : \mathsf{Un}} \qquad \frac{\Gamma \vdash M : \mathsf{Un} \quad \Gamma \vdash N : \mathcal{T} \quad \Gamma \nvdash N : \mathcal{S}}{\Gamma \vdash M : \mathsf{Dyn}(N,\mathcal{S})} \qquad \frac{\Gamma \vdash M : \mathcal{T}}{\Gamma \vdash \#(M) : \mathsf{Un}}$$

judging whether the sender is acutally well-typed. If a later recipient can establish that N does type at $S \to \mathsf{Proc}$ they can use the contents of the value safely.

Results. The main result of the type system is the following theorem of robust safety, which states that well-formed processes are robustly safe. We elide the proof, which is fairly standard and follows from lemmas for *subject reduction* (if $\Gamma \vdash P$ and $P \longrightarrow Q$ then $\Gamma \vdash Q$) and *opponent typability* (if O is an initial Δ-opponent then $\Delta \vdash O$).

THEOREM (ROBUST SAFETY). *If $\Delta \vdash P$ then P is robustly Δ-safe.*

Robust safety can be ensured, for example, by limiting interactions with opponents to untyped data communicated over untyped initial channels, however using dynamic verification one should also be able to safely accept and conditionally execute an abstraction from an opponent if the abstraction can be proven to be well typed. To this aim we internalize the type system into the language by describing it as a verifier.

Let T be a type and Δ type environment. Then $\mathcal{P}(M) = \Delta, fn(M) : \widetilde{\mathsf{Top}} \vdash M : T$ denotes a verifiable property supported by $dom(\Delta)$. A verifier then has the form: "$\mathsf{verify}\ M\ \mathsf{as}\ \mathcal{P}(x)\ \mathsf{in}\ P\ \mathsf{else}\ Q$." (Note that the addition of $fn(M) : \widetilde{\mathsf{Top}}$ to the type environment allows accepted abstractions to contain arbitrary extra free names as long as they do not affect typability.)

We are helped by the fact that the verifier is itself well-typed more or less by definition because the relevant clauses in the encoding are drawn from the set of well-typed terms, which allows us to type x at $T \to \mathsf{Proc}$. If the verification succeeds x gets bound to N in P. Since N types only at Un, the $verify$ construct implements what amounts to

WELL-FORMED PROCESSES ($\Gamma \vdash P$)

Trusted Processes:

$$\Gamma \vdash 0 \qquad \frac{\Gamma \vdash a : \mathsf{Ch}(\mathcal{T}) \quad \Gamma \vdash M : \mathcal{T} \to \mathsf{Proc}}{\Gamma \vdash a?M} \qquad \frac{\Gamma \vdash M : \mathsf{Ch}(\mathcal{T}) \quad \Gamma \vdash N : \mathcal{T}}{\Gamma \vdash M!N}$$

$$\frac{\Gamma \vdash M : \mathcal{T} \to \mathsf{Proc} \quad \Gamma \vdash N : \mathcal{T}}{\Gamma \vdash M \cdot N} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \,|\, Q} \qquad \frac{\Gamma, a : \mathcal{T} \vdash P}{\Gamma \vdash \nu a.P} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash *P}$$

$$\frac{\Gamma \vdash f(\widetilde{M}) : \mathcal{T} \quad \Gamma, x : \mathcal{T} \vdash P}{\Gamma \vdash \mathsf{let}\, x = f\langle \widetilde{M} \rangle \;\mathsf{in}\; P} \qquad \frac{\Gamma \vdash M : \mathcal{T} \times \mathcal{S} \quad \Gamma, x : \mathcal{T}, y : \mathcal{S} \vdash P}{\Gamma \vdash \mathsf{case}\; M \;\mathsf{of}\; \exists (x,y).(x,y) \;\mathsf{in}\; P}$$

$$\frac{\Gamma \vdash M : \mathsf{Dyn}(N, \mathcal{S}) \quad \Gamma \vdash N : \mathcal{S} \quad \Gamma \vdash T : \mathrm{TYPE} \quad \Gamma, x : T \vdash P}{\Gamma \vdash \mathsf{case}\; M \;\mathsf{of}\; \exists x.\mathsf{dyn}(x, T) \;\mathsf{in}\; P}$$

$$\frac{\Gamma \vdash M : \mathsf{Key}(\mathcal{T}) \quad \Gamma, x : \mathcal{T} \vdash P}{\Gamma \vdash \mathsf{case}\; M \;\mathsf{of}\; \exists x.\mathsf{enc}(x, M) \;\mathsf{in}\; P} \qquad \frac{\Gamma \vdash M : \mathcal{T} \quad \Gamma, \widetilde{y} : \widetilde{\mathsf{Top}} \vdash N : \mathcal{T} \quad \Gamma, \widetilde{y} : \widetilde{\mathsf{Top}} \vdash P}{\Gamma \vdash \mathsf{case}\; M \;\mathsf{of}\; \exists \widetilde{y}.N \;\mathsf{in}\; P}$$

Opponent Processes:

$$\frac{\Gamma \vdash a : \mathsf{Un} \quad \Gamma \vdash M : \mathsf{Un}}{\Gamma \vdash a?M} \qquad \frac{\Gamma \vdash M : \mathsf{Un} \quad \Gamma \vdash N : \mathsf{Un}}{\Gamma \vdash M!N} \qquad \frac{\Gamma \vdash M : \mathsf{Un} \quad \Gamma \vdash N : \mathsf{Un}}{\Gamma \vdash M \cdot N}$$

$$\frac{\Gamma \vdash M : \mathsf{Un} \quad \Gamma, \widetilde{x} : \widetilde{\mathsf{Un}} \vdash N : \mathsf{Un} \quad \Gamma, \widetilde{x} : \widetilde{\mathsf{Un}} \vdash P}{\Gamma \vdash \mathsf{case}\; M \;\mathsf{of}\; \exists \widetilde{x}.N \;\mathsf{in}\; P}$$

a dynamic cast, allowing one to take arbitrary data from an untyped opponent and cast it to a well-typed abstraction.

For example, suppose $\Delta \triangleq a_{net} : \mathsf{Ch}(\mathsf{Un}), b_1 : T_1, \ldots, b_n : T_n$ where $T_{1-n} \neq \mathsf{Ch}(\mathsf{Un})$ and define $\mathcal{P}(M)$ as $\Delta, \widetilde{fn(M) : \mathsf{Top}} \vdash M : (T_1 \times \ldots \times T_n) \to \mathsf{Proc}$. Then the following process is robustly Δ-safe.

$$*a_{net}?(x : \mathsf{Un})\mathsf{verify}\; x \;\mathsf{as}\; \mathcal{P}(y) \;\mathsf{in}\; (y \cdot \widetilde{b})$$

The process repeatedly reads arbitrary values from an open network channel (a_{net}) and tests them dynamically to see if they are well-typed at $(T_1 \times \ldots \times T_n) \to \mathsf{Proc}$ before applying them to a series of protected channels. If $\widetilde{b}$ represent, for example, a series of protected system calls this process could represent a virtual machine that performs bytecode verification, as well as many other applications of dynamic verification.

4 Example: Dynamic Verification and Trusted Computing

On its own, dynamic verification can be used to conditionalize the application of an abstraction on the results of static analysis of the program code. In this section we expand the use of dynamic verification to also conditionalize interactions with *running processes* using remote attestation. This solution utilizes a notion of code identity, whereby an active process is identified by the process abstraction it started as.

Background. Trusted computing is an architecture for secure distributed computing where trust is rooted in a small piece of hardware with limited resources known as the *trusted platform module (TPM)*. The TPM is positioned in the boot sequence in such a way that it is able to observe the BIOS code as it is loaded. It takes and stores the hash of the BIOS as the system boots, thus establishing itself as the root of a chain-of-trust; a secure BIOS records the hash of the operating system kernel with the TPM before it loads, and a secure operating system records the hash of an application before it is executed. If the BIOS and operating system are known to be trustworthy, then the sequence of hashes will securely identify the currently running program. Remote attestation is a protocol by which an attesting party demonstrates to a remote party what code it is currently running by having the TPM sign a message with a private key and the contents of its hash register. If the recipient trusts the TPM to identify the BIOS correctly, and knows of the programs that hash to each identity in the chain, then they can use static analysis of the program code to establish trust in the message.

Representing a Trusted Computing Platform. We represent a trusted computing framework as follows. The TPM is represented by a process parameterized on a boot channel (a_{boot}) and an attestation identity key (a_{aik}). The TPM listens on the boot channel for an operating system abstraction to load; upon receiving the OS (x_{os}) it reserves fresh attestation (b_{at}) and check (b_{chk}) channels and instantiates the OS with the new channels. This calling convention is expressed as an abstraction type for "certifiable" programs, which we abbreviate $Cert$. The TPM accepts requests on the attestation channel in the form of a message and callback channel. An attestation takes the form of a message signed by the TPMs attestation identity key where the contents are a dynamically typed message where the type is bounded by a provenance tag; that is, of the form $\mathsf{dyn}\langle y_{msg}, \mathsf{Dyn}(\#(x_{os}), \mathsf{Hash}(Cert))\rangle$. This message is then encrypted with a_{aik} and returned on the callback. The check channel is provided to clients so that they can verify TPM signatures; the TPM simply tests the signature and, if successful, returns the payload typed at Dyn.

DEFINITIONS

$$Cert \triangleq (\mathsf{Ch}(\mathsf{Dyn} \times \mathsf{Ch}(\mathsf{Un})) \times \mathsf{Ch}(\mathsf{Un} \times \mathsf{Ch}(\mathsf{Dyn})) \times \mathsf{Ch}(\mathsf{Dyn})) \to \mathsf{Proc}$$

$$\begin{aligned} TPM(a_{boot}, a_{aik}) \triangleq\ & *a_{boot}?((x_{os}, x_{arg}))\nu b_{at}.\nu b_{chk}.\big(\ x_{os} \cdot \langle b_{chk}, b_{at}, x_{arg}\rangle \\ & \mid *b_{at}?((y_{msg}, y_{rtn}))y_{rtn}!\mathsf{enc}\langle \mathsf{dyn}\langle y_{msg}, \mathsf{Dyn}\langle \#\langle x_{os}\rangle, \mathsf{Hash}(Cert)\rangle\rangle, a_{aik}\rangle \\ & \mid *b_{chk}?((z_{msg}, z_{rtn}))\mathsf{case}\ z_{msg}\ \mathsf{of}\ \mathsf{enc}(x, a_{aik})\ \mathsf{in}\ z_{rtn}!x\ \big) \end{aligned}$$

$$\begin{aligned} OS \triangleq\ & ((x_{at1}, x_{chk}, x_{arg}))\nu b_{run}.\big(x_{arg}!b_{run} \\ & \mid *b_{run}?((y_{app}, y_{arg}))\nu b_{at2}.\big(y_{app} \cdot \langle x_{chk}, b_{at2}, y_{arg}\rangle \\ & \quad \mid *b_{at2}?((y_{msg}, y_{rtn}))x_{at1}!(\mathsf{dyn}\langle y_{msg}, \mathsf{Dyn}\langle \#\langle y_{app}\rangle, \mathsf{Hash}(Cert)\rangle\rangle, y_{rtn})\ \big) \end{aligned}$$

An example of a trustworthy operating system, *OS*, is initialized with an attestation and a check channel. It repeatedly accepts outside requests to run abstractions; a fresh attestation channel is created for each request that binds a message to the identity of the abstraction before passing it on to the TPM. For user programs, such as virtual machines or Internet browsers, that themselves host outside code, this protocol can be extended arbitrarily. Each layer provides the next layer up with an attestation service that appends the clients identity to a message before passing the request down. Attestation channels are general-purpose, therefore typing the ultimate payload of an attestation requires dynamic types. The ultimate form of an attestation, then, is that of some nested series of dynamically typed messages with the innermost carrying the payload and actual type and each successive layer being of the form $\mathsf{dyn}(M, \mathsf{Dyn}(\#(N)))$ where $\#(N)$ identifies the layer that generated M. The outermost message is then signed by the TPM.

Initial Processes. We assume that initial processes have the following configuration. Execution occurs in the context of an initial environment (Δ) consisting of a fixed number of $\mathsf{Ch}(\mathsf{Top})$-typed channels ($a_1, \dots, a_j$), an arbitrary number of $\mathsf{Ch}(\mathsf{Un})$-typed channels ($a_{j+1}, \dots, a_k$) and some number of additional channels ($\widetilde{b}$) at various types. The trusted world consists of k TPM processes which share a single aik key name but listen on individual boot channels, and j subjects ($P_1, \dots, P_j$) with $fn(P_i) \subseteq \{a_i\} \cup \widetilde{b}_i$ where $\widetilde{b}_1, \dots, \widetilde{b}_j$ are disjoint subsets of $\widetilde{b}$, and a Δ-opponent (O) with $fn(O) \subseteq \{a_i \mid i > j\}$. The opponent may control any number of TPM channels, but none that are in use by another subject. No two subjects initially share a name that is not also known to the opponent, therefore any secure communications between subjects has to be brokered by the TPM.

$$\Big(\nu a_{aik}.\Pi_{i\le k}\big(TPM(a_{aik}, a_i)\big)\Big) \mid P_1 \mid \dots \mid P_j \mid O$$

A typical subject "boots up" by sending the TPM an OS file and a fresh channel. After receiving an OS callback, the subject loads some number of concurrent applications. Each application receives its own identifying attestation channel from the operating system.

$$P_i \triangleq \nu b.(a_i!(OS, b) \mid b?(x)x!(APP_1, \widetilde{b}_i) \mid \dots \mid x!(APP_k, \widetilde{b}_i))$$

The robust safety of an initial process follows from the typability of $TPM(a_{aik}, a_i)$ for all i, which we establish informally by noting that (1) when the TPM receives a well-typed OS, the new attestation channel will type at $\mathsf{Ch}(\mathsf{Dyn} \times \mathsf{Ch}(\mathsf{Un}))$, and attestations will have the form $\mathsf{dyn}(M, \mathsf{Dyn}(\#(OS), \mathsf{Hash}(Cert)))$ which will be well typed because $\Gamma \vdash M : \mathsf{Dyn}$; and (2) when the TPM loads an untyped OS, the new attestation channel will type at $\mathsf{Ch}(\mathsf{Top})$ and attestations will have the form $\mathsf{dyn}(M, \mathsf{Dyn}(\#(OS), \mathsf{Hash}(Cert)))$, which will be well typed because $\Gamma \nvdash OS : Cert$.

Using Attestations. Even a signed attestation cannot be automatically trusted. Because the opponent controls some number of TPMs, the signature provides assurance only that the message was created by a TPM that was initially running the particular abstraction that hashes to the attested identity. To trust the contents one must also trust that the attesting abstraction (1) protects its attestation channel, and (2) only generates accurate dynamic types, which in the case of nesting implies that a host program correctly identifies a hosted application when attestations are created.

Destructing an attestation is a three-step process. First the signature is validated using the b_{chk} channel provided by the TPM, which returns $\mathsf{dyn}(M, \mathsf{Dyn}(\#(OS), \mathsf{Hash}(Cert)))$. Second, the identity $\#(OS)$ is checked to ensure that it corresponds to an abstraction that types at $Cert$. Dynamic verification cannot be used here because the original program code is not recoverable from the hash, so checking the identity amounts to testing equality with something with which there is a priori trust. Attested messages will generally be nested so this process is repeated once per layer, eventually exposing a value of the form $\mathsf{dyn}(L,T)$. This is matched against an expected type and the payload L is recovered, typed at T. The processes of creating and destructing attestations are summarized in the following derived forms:

DERIVED FORMS: ATTEST AND CHECK

$$\begin{array}{lcl}
\mathsf{let}\ x = \mathit{attest}(M_{at},N,T)\ \mathsf{in}\ P & \triangleq & \nu b.M_{at}!(\mathsf{dyn}\langle N,T\rangle,b) \mid b?(x)P \\
\mathsf{let}\ x = \mathit{check}(M_{chk},N,(L_{1\ldots n}),T)\ \mathsf{in}\ P & \triangleq & \nu b.M_{chk}!(N,b) \mid \big(b?(x) \\
& & \quad \mathsf{case}\ x\ \mathsf{of}\ \mathsf{dyn}(y_1,\mathsf{Dyn}(L_1,\mathsf{Hash}(\mathit{Cert})))\ \mathsf{in} \\
& & \quad \vdots \\
& & \qquad \mathsf{case}\ y_{n-1}\ \mathsf{of}\ \mathsf{dyn}(y_n,\mathsf{Dyn}(L_n,\mathsf{Hash}(\mathit{Cert})))\ \mathsf{in} \\
& & \qquad\quad \mathsf{case}\ y_n\ \mathsf{of}\ \mathsf{dyn}(z,T)\ \mathsf{in}\ P\ \big)
\end{array}$$

The robust safety theorem, combined with the typability of the derived forms for attest and check, implies that any well-typed program written to use this infrastructure is robustly safe.

Bidirectional Authentication with a Trusted Verifier. We now turn to a specific example that uses trusted computing to allow two mutually distrusting parties to authenticate. The parties initially share no secure channels, have no knowledge of the other's program code and are unwilling to share their source code with the other. (Swapping source code may be unacceptable in practice due to proprietary interests, or simply performance reasons.) The parties do however initially trust the same verifier which together with the TPM is sufficient to establish bidirectional trust. This very general example is broad enough to suggest a wide range of applications, particularly in the context of communication over the public Internet where parties are frequently anonymous.

The example comprises three software components: TV defines a trusted third-party verifier, CLIENT defines the initiator of the communication, and SERVER defines the other party to the communication. The trusted verifier inputs an abstraction on a public channel (a_{ver}) and uses dynamic verification to test it for typability. If successful, the hash of the abstraction is taken and packed into an attestation typed at $\mathsf{Hash}(\mathit{cert})$, which is returned to the requester to be used as a certificate. CLIENT and SERVER are each passed their own abstractions when they are initialized, which they send to the verifier to obtain certificates. CLIENT initiates the communication by sending first its certificate and second an attested response channel on the public channel a_{req}. SERVER reads the certificate and uses it to trust the second message and recover the typed response channel, on which it writes its own certificate and another attestation containing the secret data.

$$\begin{array}{rcl}
\mathit{TV} & \triangleq & ((x_{at},x_{chk},\_))a_{ver}?((y_{val},y_{rtn})) \\
& & \quad \mathsf{verify}\ y_{val}\ \mathsf{as}\ \{M \mid \Gamma, fn(M) : \widetilde{\mathsf{Top}} \vdash M : \mathit{Cert}\}(z_1)\ \mathsf{in} \\
& & \quad\ \mathsf{let}\ z_2 = \mathit{attest}(x_{at},\#(z_1),\mathsf{Hash}(\mathit{Cert}))\ \mathsf{in}\ y_{rtn}!z_2 \\
\mathit{CLIENT} & \triangleq & ((x_{at},x_{chk},x_{arg}))\nu b.x_{arg}!b \mid b?((x_{self}))(\nu a.a_{ver}!(x_{self},a) \mid a?(y_{cert})a_{req}!y_{cert}) \\
& & \quad \mid (\nu b_{rsp}.\mathsf{let}\ y_{req} = \mathit{attest}(x_{at},b_{rsp},\mathsf{Ch}(\mathsf{Top}))\ \mathsf{in}\ (a_{pub}!y_{req}) \\
& & \quad\ \mid b_{rsp}?(y)\mathsf{let}\ z_{sid} = \mathit{check}(x_{chk},y,(\#(\mathit{OS}),\#(\mathit{TV})),\mathsf{Hash}(\mathit{Cert}))\ \mathsf{in} \\
& & \quad\quad b?(y)\mathsf{let}\ z_{dat} = \mathit{check}(x_{chk},y,(\#(\mathit{OS}),z_{sid}),\mathsf{Ch}(T))\ \mathsf{in}\ P)
\end{array}$$

$$\begin{aligned} \mathit{SERVER} \triangleq\ & ((x_{at}, x_{chk}, x_{arg}))\nu b.x_{arg}!b \mid b?((x_{self}, x_{dat}))\nu a.a_{ver}!(x_{self}, a) \mid a?(y_{cert}) \\ & a_{pub}?(y)\mathsf{let}\ y_{cid} = \mathit{check}(x_{chk}, y, (\#(\mathit{OS}), \#(\mathit{TV})), \mathsf{Hash}(\mathit{Cert}))\ \mathsf{in} \\ & a_{pub}?(y)\mathsf{let}\ y_{req} = \mathit{check}(x_{chk}, y, (\#(\mathit{OS}), y_{cid}), \mathsf{Ch}(\mathsf{Top}))\ \mathsf{in} \\ & (y_{req}!y_{cert}) \mid \mathsf{let}\ y_{resp} = \mathit{attest}(x_{chk}, x_{dat}, T)\ \mathsf{in}\ (y_{req}!y_{resp} \mid Q) \end{aligned}$$

We assume that all three components will be run on trusted platforms with CLIENT and SERVER on distinct TPMs. Trust in the verifier is based on the identity of the program code, not the party running it, therefore it can be run on its own TPM, or on the same TPM as either party, or even as separate processes on both. The TPM therefore allows parties to reliably certify their own code.

5 Conclusions

We have presented a new *reflective* variant of the higher-order π calculus that allows for the dynamic inspection of process syntax and is useful for modeling open systems, which often rely on such operations. Reflection has also been considered for the λ-calculus [23, 24], and dynamic verification using type-checking primitives has been considered in a π-calculus [25]. Allowing complete observation of process syntax in a higher-order π-calculus, however, appears to be novel, noting concurrent work by Sato and Sumii [7].

We considered two specific applications that use reflection: *dynamic verification*, which relies on an ability to dynamically typecheck mobile code prior to execution, and *trusted computing*, which relies on an ability to associate a running process with the identity of the process abstraction it started as.

The genesis of this work was our previous work with trusted computing in higher-order pi [6]. Many issues, such as code identity and allowing attackers to extract names from mobile code, were considered in the previous paper but handled in an ad-hoc fashion. This paper fulfills two additional objectives. First, it comprises a more foundational and expressive approach to understanding such systems. Second, it has allowed us to internalize static analysis. The approach to trusted computing in this paper lacks rich access control features which were the focus of the prior paper, however adding them would not be difficult.

References

1. Yellin, F.: Low-level security in Java. In: WWW4 Conference. (1995)
2. Necula, G.C.: Proof-carrying code. In: Principles of Programming Languages (POPL '97). (1997)
3. Riely, J., Hennessy, M.: Trust and partial typing in open systems of mobile agents. In: Principles of Programming Languages (POPL 99). (1999)
4. Trusted Computing Group http://www.trustedcomputinggroup.org: TCG TPM Specification Version 1.2. (March 2006)
5. Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Computer and Communications Security (CCS), New York, NY, USA, ACM Press (2004) 132–145
6. Cirillo, A., Riely, J.: Access control based on code identity for open distributed systems. In: Trustworthy Global Computing, Springer-Verlag (November 2007)

7. Sato, N., Sumii, E.: A higher-order, call-by-value applied pi-calculus. In: Seventh Asian Symposium on Programming Languages and Systems (APLAS 2009), Springer-Verlag (2009)
8. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD thesis, University of Edinburgh (1993)
9. Sangiorgi, D.: Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). Theoretical Computer Science **253** (2001) 311–350
10. Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press (2001)
11. Sun Microsystems: Java Object Serialization Specification. (2005) Available at http://java.sun.com/javase/6/docs/platform/serialization/spec/serialTOC.html.
12. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification Second Edition. Sun Microsystems (1999)
13. Anderson, N.: Hacking Digital Rights Management. ArsTechnica.com. (July 2006) http://arstechnica.com/articles/culture/drmhacks.ars.
14. 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)
15. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Principles of Programming Languages (POPL '01). (2001)
16. Abadi, M., Gordon, A.: A calculus for cryptographic protocols: The spi calculus. In: Information and Computation. Volume 148. (1999) 1 to 70
17. Abadi, M.: Secrecy by typing in security protocols. J. ACM **46**(5) (1999)
18. Gordon, A.D., Jeffrey, A.S.A.: Authenticity by typing for security protocols. J. Computer Security **11**(4) (2003)
19. Fournet, C., Gordon, A., Maffeis, S.: A type discipline for authorization policies. In: ESOP '05. (2005)
20. Gordon, A.D., Jeffrey, A.S.A.: Secrecy despite compromise: Types, cryptography, and the pi-calculus. In: CONCUR. (2005)
21. Fournet, C., Gordon, A., Maffeis, S.: A type discipline for authorization in distributed systems. CSF **00** (2007) 31–48
22. Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically typed language. ACM Transactions on Programming Languages and Systems **13**(2) (1991) 237–268
23. Alt, J., Artemov, S.: Reflective lambda-calculus. Proof Theory in Computer Science (2001) 22 – 37
24. Artemov, S., Bonelli, E.: The intensional lambda calculus. Logical Foundations of Computer Science (2007) 12 – 25
25. Maffeis, S., Abadi, M., Fournet, C., Gordon, A.D.: Code-carrying authorization. In: ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security, Berlin, Heidelberg, Springer-Verlag (2008) 563–579