Trusted Computing, Trusted Third Parties, and Verified Communications

Martín Abadi
University of California at Santa Cruz

Abstract

Trusted Computing gives rise to a new supply of trusted third parties on which distributed systems can potentially rely. They are the secure system components (hardware and software) built into nodes with Trusted Computing capabilities. These trusted third parties may be used for supporting communications in distributed systems. In particular, a trusted third party can check and certify the data sent from a node A to a node B, so that B can have some confidence in the properties of the data despite A's possible incompetence or malice. We present and explore this application of Trusted Computing, both in general and in specific instantiations.

1 Introduction

Trusted third parties can be useful in a variety of tasks in distributed systems. For instance, certification authorities are helpful in associating public keys with the names of users and other principals; in multi-player games, servers can contribute to preventing some forms of cheating; and smart-cards with limited resources may rely on trusted, off-card servers for verifying downloaded bytecode class files. Unfortunately, resorting to trusted third parties is not always practical, as it typically results in deployment difficulties, communication overhead, and other costs. Moreover, well-founded trust is scarce in large-scale distributed systems, and so are reliable trusted third parties.

This paper considers new trusted third parties that may appear within general-purpose computing platforms as a result of several current efforts. Those efforts include substantial projects in industry, such as the work of the former Trusted Computing Platform Alliance (TCPA) and its successor the Trusted Computing Group (TCG), and Microsoft's Next Generation Secure Computing Base (NGSCB, formerly known as Palladium) [11]. They also include research projects such as XOM [19] and Terra [13]. The trusted third parties are the secure system components (hardware and software) built into nodes with Trusted Computing capabilities.

These trusted third parties can contribute to both secrecy and integrity properties in distributed systems. In particular, when two nodes A and B communicate, the trusted third party embedded in A can check and certify the messages that A sends to B. This verification may have a variety of meanings—it can for example ensure the well-formedness of data fields, the absence of known viruses, the safety of mobile code, or the validity of certificate chains. The verification can offer security guarantees to B, often more efficiently than if B performed the check itself. Although the verification clearly depends on A's secure system components, it is protected against malfunctions in the rest of A, and can prevent their spread to B. The description and study of this scenario are the main contents of this paper.

The next section discusses efforts such as TCPA, the appearance of new trusted third parties, and (briefly) the applications that they may enable. Section 3 sets out our assumptions. Section 4 explains the use of a trusted third party for verified communications, on a case-by-case basis. Section 5 considers some examples, and section 6 summarizes benefits and drawbacks. Section 7 then outlines more general mechanisms for verified communications; it relies on machinery for remote invocation and on extensible runtimes. Section 8 develops an example. Section 9 discusses extensions in which data is partly secret or generated by the trusted third party. Section 10 concludes.

2 New trusted third parties?

Next we identify more precisely the new third parties described in the introduction, and consider whether they should be trusted. We also discuss the applications (some old, some new) that may rely on this trust.

2.1 The new third party

With systems such as NGSCB, a computing platform includes a protected execution environment, with protected memory, storage, and I/O. The platform is open in that it can run arbitrary programs like today's ordinary PCs, but those arbitrary programs should not compromise the security kernel or any subsystem under its protection. Moreover, the security kernel can authenticate the programs, and it in turn can be remotely authenticated.

Therefore, the security kernel may serve as a trusted third party for an interaction in a distributed system. Conveniently, this trusted third party is local to a node. In particular, the security kernel may assist a remote principal in interactions with the rest of the node, which may be arbitrarily corrupted. Moreover, the security kernel may communicate directly with a local human user, through secure I/O; it may therefore assist the user in its interactions with the rest of the node.

A subsystem protected by the security kernel may also play the role of trusted third party. Through standard delegation techniques (e.g., [17]), the protected subsystem can act on behalf of the security kernel and its clients. The main advantage of relying on a protected subsystem is to retain, to the extent possible, the simplicity, manageability, and security of the kernel proper.

Figure 1 is a typical picture of a system with NGSCB. It shows a system with two sides: a left-hand side with arbitrary software (not necessarily trusted) and a right-hand side with secure system components, including an operating system and user-mode code.

2.2 Applications

This trusted third party can contribute to security in distributed systems, in several ways:

- The trusted third party can contribute to secrecy properties, for example holding secrets for a user, and presenting those secrets only to appropriate remote servers. The secrets would be kept from viruses that may come with arbitrary programs.

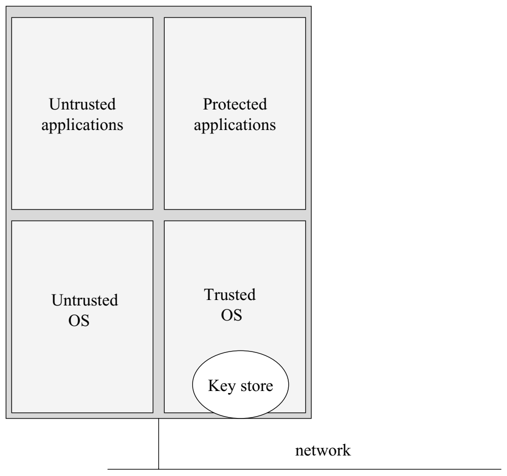

Figure 1: A typical picture of a system with NGSCB

- The trusted third party can also contribute to integrity properties, for example checking incoming and outgoing data. In particular, as suggested in the introduction and explained in section 4, the trusted third party embedded in a node A can check and certify the messages that A sends to another node B. The trusted third party can protect B against A's incompetence or malice, for example against A's viruses.

While the secrecy properties have received a fair amount of attention, we believe that the opportunities and problems related to integrity are also important. They are the focus of this paper.

One may wonder also about availability properties—for example, asking whether the trusted third party can help protect against denial-of-service attacks. We address availability only indirectly (see section 6).

Trusted Computing is often narrowly associated with protecting movies and other proprietary content on commodity platforms, but it enables other significant applications. Several of those applications remain in the broad realm of Digital Rights Management (DRM). For instance, users may want to attach rights restrictions to their e-mail messages and documents; protected execution environments can help in enforcing those restrictions. Similarly, however, it has been argued that protected execution environments enable censorship and other worrisome applications [2]. Beyond DRM, NGSCB could be employed for secure document signing and transaction authorization [11], for instance. Notwithstanding such intriguing ideas, it appears that the thinking about applications remains active, and far from complete. One of the goals of this paper is to contribute to this thinking.

2.3 Limits on trust

TCPA, TCG, and NGSCB have been rather controversial. While they are associated with the phrases "Trusted Computing" or "Trustworthy Computing", they have also been called "Treacherous Computing" [25]. Relying on them in the manner described in this paper will perhaps be considered naive. Even putting aside any consideration of treachery, trust should not be absolute, but relative to a set of properties or actions, and it is dangerous to confuse trusted and trustworthy.

Following Anderson [1], we mostly use an acronym rather than "Trusted Computing" or a similar name. We pick SCB, which may stand for "Secure Computing Base" (or "Sneaky Computing Base") because the descriptions in this paper focus on NGSCB, as we explain in section 3. By an SCB we loosely mean a collection of system components, hardware and software, including a security coprocessor with cryptographic keys and capabilities, a microkernel or other operating system, and possibly some protected subsystems running on top of these. Section 3 lists our assumptions more specifically.

The trust that one places on an SCB may be partly based on the properties of its hardware. If this hardware is easy to subvert, then assurances by the SCB may be worthless. On the other hand, a modest level of tamper-resistance may be both achievable and sufficient for many applications. First, attacks on hardware (unlike buffer-overflow attacks, for instance) are not in general subject to large-scale automation. Moreover, many nodes (and their SCBs) are in physical environments in which serious tampering is hard or would be easily detected—for example, in shared workspaces and data centers. In other environments, a key question is whether the people who are in a position to perform the tampering would benefit from it. Whenever the SCB works on behalf of users, defending them from viruses and other software attacks, we may not need to worry about protecting the SCB from the users.

Trust in an SCB may also be partly based on trust in its developer, its administrators, and other principals. For instance, if Acme makes chips with embedded secret keys, and issues certificates for the corresponding public keys, then the chips are reasonable trusted third parties only if Acme can be trusted to manage the secret keys appropriately. Thus, Acme is a trusted third party too. However, trust in Acme may be based on an open review, and may be further justified if Acme never has direct access to the secret keys.

On this basis, it seems reasonable or at least plausible that SCBs would be trusted third parties—and even trustworthy third parties—in specific contexts.

3 Assumptions

We focus on NGSCB partly because of its practical importance, partly for the sake of concreteness, but most of the paper applies verbatim to other systems such as XOM; it may also apply to future versions of these systems, which continue to evolve. This section presents the main assumptions on which we rely.

We expect that the SCB in a system is able to communicate with other parts of the system, typically at a modest cost; in particular, this communication may be through local memory. In addition, we make the following assumptions:

- Authenticity: The capability of making assertions that can be verified by others (local or remote) as coming from this SCB, or from an SCB in a particular group. For instance, in a common design, the SCB holds a signature key that it can use for signing statements; a certification authority (perhaps operated by the SCB's manufacturer, owner, or a delegate) issues certificates for the corresponding public key, associating the public key with this SCB or with a group of trusted SCBs.

- Protection: Protection from interference from the rest of the system when performing local computations.

Two additional assumptions are not essential, but sometimes convenient:

- Persistent state: The SCB may keep some persistent state across runs. This state may be as simple as a monotonic counter. Using this monotonic counter, the SCB may implement mechanisms for maintaining more complex state. In particular, assuming that the SCB has a monotonic counter, it can maintain other state on untrusted storage, using digital signatures and encryption; the counter should be incremented, and its value attached to the state, whenever an update happens, thus offering protection against replay attacks.

- Weak timeliness: The SCB has secure means to know the time, to within the precision allowed by network and scheduling delays. In particular, the SCB may get the correct time signed by a trusted network time server TS for which it knows the public key. In each exchange with TS, the SCB would challenge TS with a fresh nonce (for example by applying a one-way hash function to a secret plus a monotonic counter). Network and scheduling delays may lead the SCB to accept an old value for the time, but never a future value.

  Without this assumption, the SCB can include nonces as proofs of timeliness for its assertions to on-line interlocutors. The nonces would be provided as challenges by those interlocutors. The assumption removes the need for the challenge messages.

On the other hand, we do not assume several properties that may be hard to obtain.

- No availability: If the SCB never runs, or runs with degraded performance, then the availability of the whole system may suffer, but with no direct effect on integrity or secrecy properties. The SCB may not have a reliable way to detect when it is running slowly.

  Since we do not assume availability, we cannot assume synchronized clocks. The SCB may try to verify the timeliness of incoming statements (using nonces or timestamps), but a scheduling delay may compromise the accuracy of this verification.

- No obfuscation: All elements of the SCB design and implementation can be open; only its secret signature key needs to remain secret.

- No secure I/O: There is no requirement of secure communication with keyboards, mice, display, network ports, or other external entities, beyond what is given by the capability of signing and checking assertions.

Availability is difficult to achieve in part because it requires control of scheduling. Obfuscation has often been tried, for example in DRM systems; open design is typically deemed superior, particularly for systems that should be trusted. Secure I/O is central to some of the envisioned applications of SCBs, but it requires support that we cannot universally take for granted at present, and it also could benefit from progress on user interfaces.

4 Verified communications with an SCB

In this section we show how an SCB can serve as a trusted third party for checking and certifying communications. First, in section 4.1, we review examples of input verification, and their importance for security. Then, in section 4.2, we explain how these examples can rely on SCB support. Later sections are concerned with refining the examples, discussing benefits and drawbacks, and generalizing. Throughout the paper, we emphasize communications that involve programs at their endpoints. Accordingly, we often refer to the sender as the caller and to the receiver as the callee. However, many of the ideas and techniques that we present do not require that the messages being exchanged are calls to program functions; they apply more broadly to arbitrary messages in a distributed setting.

4.1 Checking inputs

When a program receives data, it is prudent that it verify that the data has the expected properties before doing further computation with it (e.g., [15]). These verifications may for example include:

- Checking that an argument is of the expected size, thus thwarting buffer-overflow attacks.
- Checking that a graph is acyclic, so as to avoid infinite loops in later graph manipulations.
- Checking that an argument is of the expected type or structure.
- Checking the validity of a "proof of work" (evidence that the sender has performed some moderately hard computation, of the kind suggested for discouraging spam; e.g., [10, 16]).
- Checking that cryptographic parameters have particular properties (often number-theoretic properties) needed for security [3, Principle 6].
- Checking that a set of credentials forms a chain and implies some expected conclusion, for example that the sender is a member of a group.

Further, interesting examples arise in cases where the data is code (or may include code):

- Checking that the data does not contain one of a set of known viruses.
- Checking that a piece of mobile code is well-typed. This mobile code might be written in a source language, an intermediate language, or in binary. As in Java Virtual Machines [20] and the Common Language Runtime (CLR) [6], the typing provides a base level of security. With some research type systems (e.g., [8, 21]), the typing may ensure further properties, such as compliance with resource-usage rules and secure information-flow properties.
- Checking the legality of a logical proof that a piece of mobile code satisfies some property, for example an application-specific safety property, termination, or an information-flow property. Research on proof-carrying code [22] explores these ideas.

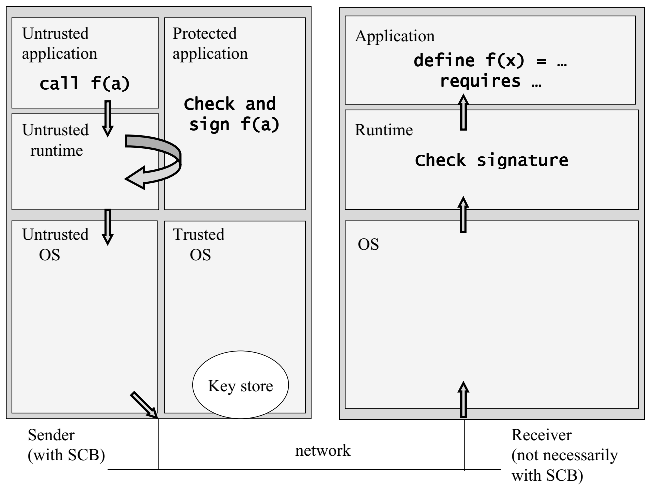

Figure 2: A verified input

- More speculatively, checking that compiled mobile code is a correct implementation of a given source program (that is, that the compiler did not make a mistake in a particular run). Research on translation validation [24] explores these ideas.

As these and other examples illustrate, authenticating the origin of data is often essential, but further checking can be essential too. In particular, the checking can serve in preventing the spread of infections from senders to receivers.

Some checking may be done automatically by standard machinery in distributed systems; for example, remote procedure call (RPC) machinery can enforce simple typing properties before delivering arguments to remotely invoked procedures. Such automatic checking is particularly justified for generic properties that are easy to verify. On the other hand, application-specific properties and properties that are expensive to verify tend to be treated on a case-by-case basis.

4.2 Using an SCB

Suppose that a piece of code relies on a certain property of its inputs, and that therefore this property should be checked. The checking can happen at the code's boundary or deeper inside the code. It could also happen at the caller, though in general the caller may not know what property to ensure, and crucially the caller cannot always be trusted.

Having an SCB in the caller leads to a new possibility, depicted in Figure 2: the SCB can serve as a trusted third party that is responsible for the checking, and that certifies that the checking has succeeded. This certification consists in a signed assertion that the call (including its arguments) satisfies a given property. The signed assertion should contain a proof of timeliness, such as a timestamp or a nonce. The signature may simply be a public-key digital signature. When the

SCB and the consumer of the signature share a secret, on the other hand, the signature may be an inexpensive MAC (message authentication code). This MAC may be applied automatically if caller and callee communicate over an authenticated channel, such as can be implemented on top of the SSL and SSH protocols. This authenticated channel has another clear benefit: proving the identity of the caller to the callee.

When it receives a certificate, the callee should check that it matches the call, that it is timely, that it claims the expected property, and also that it is issued by a sufficiently trusted SCB. All these checks but the last should be straightforward. Checking that the certificate is issued by an appropriate SCB is a classical authorization problem (a matter of trust rather than of remote integrity verification). When the SCB is identified with a public key, the public key may be in a group of keys trusted for the purpose. On the other hand, the SCB may prove only that it is a proper SCB in a certain group, without revealing its exact identity; this case is more elaborate but does not introduce new difficulties.

There is no requirement that the callee have an SCB. However, an SCB at the callee can provide a secure environment in which to perform the checks just described; it can also serve for certifying properties of communications in the opposite direction, such as the result (if any) of the call.

There remains the problem of letting the caller's SCB know what property to check. This information may be hard-wired on a case-by-case basis. In general, it is attractive to envision that the property would be advertised along with the interface to the code being called. Much like the caller learns about the existence of the code entry point, and about the expected types and semantics of arguments, the caller should learn about the expected properties of these arguments. We return to this subject below.

Using an SCB for checking inputs has a number of desirable features, as well as some potentially problematic ones. Before we discuss them, however, it is useful to consider a few instantiations of the method for particular checks.

5 Examples

Next we consider four examples, both because of their intrinsic interest and in order to elucidate general features of the method described in section 4.2.

5.1 Typechecking

In the simplest example, the SCB of the caller typechecks the call, and writes a corresponding certificate.

For simple typing properties of small arguments, this example is wasteful. If the caller's SCB and the callee are not already communicating on an authenticated channel, then the callee may need to check some public-key certificates; when typechecking is simple and fast, trading it for a public-key operation is hardly attractive.

As arguments get larger, delegating the typechecking to the caller's SCB becomes more reasonable. For instance, suppose that the caller is uploading a large amount of data into the callee's database, and that this data is supposed to be in a particular format. In general, checking or imposing this format may require some processing and some buffering. If the caller's SCB can guarantee that the format is obeyed, then the callee may need to compute a message digest (relatively fast) and perform at most one public-key operation, independently of the size of the data, without any buffering.

Delegating the typechecking to the caller's SCB also becomes more reasonable for complex typing tasks. For instance, the callee may be relieved to avoid the task of checking that a piece of XML conforms to a particular schema, or that a piece of mobile code is well-typed. Indeed, the typechecking of mobile code can be fairly expensive, to the point where it is difficult or impossible on resource-constrained environments.

In a recent paper [18], Leroy discusses the cost of traditional bytecode verification on Java cards, and also discusses alternatives. Leroy writes:

> bytecode verification as it is done for Web applets is a complex and expensive process, requiring large amounts of working memory, and therefore believed to be impossible to implement on a smart card.

The alternatives include both off-card verification and the combination of off-card code transformations with easier on-card verification. Leroy ingeniously develops this latter alternative. On the former alternative, Leroy writes:

> The drawback of this approach is to extend the trusted computing base to include off-card components. The cryptographic signature also raises delicate practical issues (how to deploy the signature keys?) and legal issues (who takes liability for a buggy applet produced by faulty off-card tools?).

Having the off-card verification done in the caller's SCB mitigates these concerns:

- Extending the trusted computing base to an SCB appears less problematic than extending it to an arbitrary machine with arbitrary software and arbitrary viruses.
- The deployment of SCBs should include the deployment of their keys and of certificates for those keys.
- The off-card verifier can be chosen by the consumer of the code, or a delegate, and the SCB can guarantee that it is this verifier that it runs. Therefore, the SCB would not be liable for a faulty verifier. (However, other parties would still have to be responsible for more fundamental infrastructure failures such as bugs in SCBs or leak of the master secret keys.)

Moreover, any work done in the caller's SCB needs to be done only once, while work done at the consumer needs to take place once per consumer (and even more often when consumers obliviously download the same piece of mobile code multiple times).

In addition to smart-cards, servers can also be resource constrained. In the design of busy servers that deal with many clients, one typically shifts as much work as possible to the clients. In our case, the client's SCB would be responsible for checking code uploaded to the server (servlets). For instance, when the server is a database, and its data cannot be sent to the client because of privacy considerations or sheer size, the client may upload code to run against the data; the client's

SCB could ensure the safety of the code. More broadly, the client's SCB could also ensure that the code conforms to any server policies.

In short, although there exist clever alternatives, typechecking in the caller's SCB appears as a viable approach to an actual problem. Although it is not always advantageous, it does have some appealing properties, and it can be a good choice.

5.2 Proof checking

Research on proof-carrying code develops the idea that mobile code should be accompanied by proofs that establish that the code satisfies logical properties. As a special case, the properties may represent basic guarantees such as memory-safety, which can also be obtained by typechecking. However, proof-carrying code is considerably more general. As suggested above, the properties may include application-specific safety properties, termination, and information-flow security properties. For example, a proof may guarantee that the code uses only certain limited resources, or that it does not leak pieces of private user data. Such properties may be attractive whether the receiver of the code is a resource-constrained personal smart-card or a busy database server.

Although the verification of proofs is typically simpler than their construction, it is not a trivial task. It is roughly as hard as typechecking (discussed in section 5.1), and in fact proof checking can be formulated as a kind of typechecking. In addition, proofs can be bulky, creating communication overhead. For example, a recent paper [14] that treats device-driver properties includes proof sizes, for instance up to 156 KB of proof for a program of around 17 KLOC. Other proof encodings are possible (e.g., [23]), and may lead to a reduction of proof sizes by an order of magnitude. While these encodings are both insightful and effective, they can lead to slower proof checking, and in any case the proofs often remain much larger than signed statements. For example, a proof for the hotjava code takes 354 KB [23], substantially less than the code itself (2.75 MB), but more than a thousand times the size of a signature; checking the proof took close to one minute on a 400 MHz machine, much more than checking a signed statement.

Alternatively, with our approach, the SCB of the code producer could be responsible for checking the proof. The proof could be constructed outside the SCB, by whatever means, and given to the SCB with a cheap, local memory transfer, rather than network communication. The SCB could then transmit an assertion that the proof exists, in a certificate, rather than the proof itself. The consumer of the code would simply check the certificate rather than the proof. Leroy's concerns about off-card bytecode verification apply also to this scenario, though again the use of an SCB should mitigate them and offer some advantages.

To date, there is only limited experience in the deployment and use of proof-carrying code technology. Therefore any assessment of the use of SCBs in this context may remain rather speculative. Nevertheless, as for typechecking, this use of SCBs appears as a sensible and potentially attractive variant.

5.3 Certificate checking

For access control in distributed systems, the reference monitor that evaluates a request typically needs to consider digitally signed certificates and assemble evidence on whether the request should be granted. If the request comes from a source S and it is for an operation O on a target object T, the certificates may for example say that S is a member of a group G, that G is included in another group G', that all members of G' can perform O on objects owned by a principal P, and that P does in fact own T. Examples with chains of 5–6 certificates are not uncommon in some systems (e.g., [7, 9]). The certificates may be obtained by a variety of methods (pushed or pulled); selecting the relevant certificates and assembling them into a proof can be difficult. Therefore, several systems have, to various extents, shifted the work of providing proofs to the sources of requests [26, 4, 5]. Nevertheless, the checking of proofs remains in the reference monitor.

Using an SCB, we can go further: the source of a request need not present a pile of certificates or even a proof, but rather its SCB can provide a certificate that it has checked a proof. (In addition, the SCB should present certificates to establish its trustworthiness, and the reference monitor should check them, but these certificates may be trivial, and in any case they should not vary much from request to request.) Thus, the task of the reference monitor becomes simpler.

This approach could also have privacy advantages: the source's SCB need not reveal all the source's certificates—including the exact identity of the source and its group memberships—as those are processed locally. Private information about the source can thus be kept from the reference monitor, and also from any parties that somehow succeed in compromising the reference monitor, which may not have an SCB. Conversely, the reference monitor may be able to disclose its access-control policy to the source's SCB without making it public. (However, this disclosure is not essential: the SCB may provide only a partial proof if it does not know the access-control policy, so the approach applies in that case also.) Clearly, realizing this privacy advantage may require additional machinery, such as specifications of privacy properties that control the flow of certificates; the development of this machinery is perhaps interesting but beyond the scope of this paper.

While the explanation above concerns a reference monitor that evaluates a request, much the same applies to an on-line authority that issues certificates—for example, an authority that issues a certificate of membership in a group G to anyone who proves membership in another group G'.

More generally, the protected environment of an SCB appears as an appealing place for certificate processing and manufacturing. With some care, its weak timeliness properties should be adequate for this application.

5.4 Virus confinement and communications censorship?

Preventing the spread of viruses is an eminently worthy application of SCBs. Because viruses can in general attack anti-virus software, it is attractive to run that software under the protection of SCBs. In particular, when two nodes communicate, either or both can use their SCBs to check and certify the absence of known viruses in the data they exchange.

One may ask, however, whether any negative applications of SCBs might make them unattractive overall. In particular, the same infrastructure that blocks viruses could well be used for censoring other kinds of contents. Fortunately, communications censorship—at least in the form described here—can be avoided. First, there may be legal protections against it. Hardware attacks on SCBs may also defeat censorship, though they negate protection against viruses at the same time. Finally, censorship may be avoided at the software level, since communications between consenting nodes can circumvent SCBs. (We note however that there has been prior discussion of other forms of censorship, in which local files would be deleted [2].)

6 Assessment

In light of the preceding examples, we see that the shift of checking to the sender's SCB has a number of consequences, some of them rather attractive:

- The work is done at the sender, not the receiver. Therefore, we may not mind if there is quite a lot of work. In particular, we remove one opportunity for denial-of-service attacks on the receiver. This point is only significant if the work is substantial (more expensive than whatever signature verification is required). It may be particularly significant when the receiver is a resource-constrained device such as a smart-card or a server.
- Any auxiliary data needed for the checking is communicated only locally, not to the receiver across a network. This feature can result in simplifications and efficiency gains (as in the proof-carrying code example), and possibly also in privacy gains (as in the certificate-checking example).
- If the data is sent to multiple destinations, the checking of each property needs to be done only once at the sender, not once at every destination. (For example, the data might be mobile code being widely distributed, as discussed above.)
- The receiver should trust the sender's SCB. Specifically, if that SCB is somehow compromised (say, with a hardware attack), the checking may be circumvented. On the other hand, the receiver need not trust the rest of the sender, which may be incompetent, compromised, or malicious.

Some of these features are also obtained when the checking is done by a trusted third party placed at a firewall or at another machine managed by trusted system administrators. In comparison, using an SCB may increase concerns about hardware attacks. On the other hand, it may reduce any concerns about administrators, it saves communication, and it does not require special infrastructure.

7 General mechanisms

It is possible to use an SCB for checking inputs on a case-by-case basis, by relying on a few specific checkers. For example, a particular trusted typechecker may run as a protected subsystem on top an SCB, and the SCB could certify that it has been successfully applied to any mobile code being sent to other hosts. The other hosts would have to know that this typechecker provides the guarantees that that they require. Such knowledge could perhaps be based on a certificate, signed by a trusted authority, that asserts that the typechecker implements a particular type discipline, and that the type discipline ensures memory-safety.

It is also possible to generalize—to provide a general mechanism whereby a callee can specify fairly arbitrary checks to be executed at its caller's SCB. In this section we outline such a general mechanism.

A general mechanism is appealing for several reasons. First, one may not be able to guess all the checks that will be wanted over time. Moreover, many of the checks that one may define and implement on a case-by-case basis could benefit from common concepts and infrastructure. Secondarily, the development of this mechanism could itself be instructive, giving some interesting experience with SCBs and other important machinery. Of course, analogous arguments can be made (and have been made) for other pieces of programming infrastructure, such as general facilities for remote invocation.

We consider verified communications in the context of systems for remote invocation. Here we use the phrase "remote invocation" rather broadly, covering the classic concepts of remote procedure call (RPC) and remote method invocation (RMI), SOAP, and also web services. There is no requirement that the caller blocks waiting for a result from an invocation, or even that the invocation has a result. We assume that, in both the caller and the callee, stub code deals with conversions between local and network formats (marshaling and unmarshaling), and more broadly hides details of network communication from application code. For instance, in RMI, the caller stub supports the illusion that the target of the invocations is a local object, forwarding invocations on a local surrogate object to the remote callee; the callee's stub dispatches the invocations to the appropriate targets. As in much of the literature, we refer to the caller as the client and to the callee as the server, but we do not assume that the server is more powerful, trusted, or shared, nor that the client is a device under direct user control.

As we outline it here, a general mechanism for verified communications deals with server set-up (how verification is specified), client set-up (how the corresponding code is put in place), client checking (how checking and certification takes place at the client's SCB), and finally server checking (how the server handles the resulting certificates):

Server set-up

- The server specifies what checks the client has to perform on arguments to calls. We may think of this as an extension of the types given in the server's interface.

- The checking may be specified with a concrete piece of checking code, or with an unambiguous, compact identifier for the checking code, such as a hash of its binary. It may also be specified more abstractly (as a logical property or otherwise); additional certificates may then assert that particular pieces of code implement the check. Thus the checking code may be different across platforms and may change over time.

  Fairly elaborate mechanisms for naming code are a probable component of SCBs in any case, for attestation [17, 11]. No new such mechanisms may be needed for our purposes.

Client set-up

- When a client learns about an interface for remote invocation, it also obtains a specification of the check to be performed. The client can then find code that implements the check (perhaps directly from the specification, perhaps from the Web) in such a way that the code will run under the protection of the client's SCB. The code may be located and installed immediately or lazily.

- The code in question may need access to libraries and other shared resources, yet the client may not be willing to run arbitrary code without protecting itself. It is therefore appealing to place the code in a safe execution environment, for example a version of the Common Language Runtime (CLR) running on top of the client's SCB. This runtime provides basic safety guarantees; it also provides a general execution environment with configurable security policies.

  A simple, straightforward sandbox suffices for most of the examples above. More configuration of security policy can be required for elaborate examples. For instance, the checking code may wish to put some state into temporary files; the security policy could allow some file access. On the other hand, the client may want to protect against alleged checking code that reads all its files and sends their contents to the server. Furthermore, the client may not be willing to run some code at all, for instance if the code is of dubious origin.

Client checking

- When the client makes a remote call, the client stub automatically invokes the SCB, passing it the call and a reference to the checking code to be run. The SCB performs the check and signs the call and the result of the check (along with a proof of timeliness), and returns the signed statement to the client stub. The client stub then sends the call across the network, along with the SCB's signed statement, and any additional certificates. Those certificates may pertain to the checking code, as mentioned above, or to the identity and quality of the SCB.

  In the case in which communication is based on XML (as in SOAP), we may think of the SCB as performing a transform on the original call. This perspective may be useful because there are already standard mechanisms for XML transforms.

- The checking code may occasionally expect additional inputs for its work. For example, the checking code may require the client stub to provide a proof. This proof is not an argument to the call to be passed across the network, and it need not be signed. Its only role is in allowing the client's SCB to assert the existence of a proof without having to construct one. More generally, the checking code may interact with the client stub and even with other entities.

Server checking

- The server stub verifies the required certificates before delivering calls to higher-level code. The verification is driven by the specification of the checks to be performed, plus information on which SCBs are trusted.

- The verification may take place in the server's SCB. Placing the verification in an SCB may provide protection against viruses in the server, attractively, but it is not essential.

In the next section we illustrate the use of this general machinery by going through an example, step by step. However, we have yet to implement the techniques of this paper in the concrete context of a system.

8 An example, step by step

We start with a simple version of the example, add features, and then consider its operation.

Suppose that a server offers a generic computing service, initially with the following interface:

```
public void compute(f : Code,
                    i : FileName,
                    o : FileName)
```

Here `f` is code to be executed (possibly in binary format), `i` a source of inputs for the code, and `o` a destination for the outputs. For security, the server may check that it is safe to run `f`, somehow—for example, by checking `f` for known viruses and also by relying on any types and other evidence of safety included with `f`. The server may also require that its clients identify themselves. Internally, it may then check their identities against access control lists, for example those for `i` and `o`. The identity of the client can be added as an argument:

```
public void compute(p : Principal,
                    f : Code,
                    i : FileName,
                    o : FileName)
```

The principal `p` is the source of the request; the secure-communication machinery can guarantee that this argument is not spoofed [17].

With our approach, the interface may specify the requirements of the call, leaving their verification to the caller's SCB:

```
public void compute(p : Principal,
                    f : Code,
                    i : FileName,
                    o : FileName)
requires     p says safe(f),
             p says may-read(p,i),
             p says may-write(p,o),
             GoodSCB(p)
```

For simplicity, this interface identifies the principal `p` with its SCB. It is however easy to write versions in which the SCB need not put its full authority behind the call, in particular by requiring only that `p` be of the form "`s quoting r`" [17], for some SCB `s` and some identity `r`:

```
public void compute(p : Principal,
                    f : Code,
                    i : FileName,
                    o : FileName)
requires for some r, s.
             p = (s quoting r),
             GoodSCB(s),
             s says safe(f),
             s says may-read(p,i),
             s says may-write(p,o)
```

Such requirements are particularly appropriate when `r` represents a piece of code at the client. Even when the SCB and its user are trustworthy (so in particular the user does not attempt hardware attacks on the SCB), some client code may not be.

When a client `p` imports this interface, it also learns about the requirements that calls should satisfy. When the client wishes to call `compute(p,f,i,o)`, it somehow finds proofs of `safe(f)`, `may-read(p,i)`, and `may-write(p,o)`. The proof of `safe(f)` may consist of a logical proof of some property of `f` and a certificate that associates the predicate name "`safe`" with this property. The proofs of `may-read(p,i)` and `may-write(p,o)` may be assertions signed by a trusted authority, perhaps by the server itself. In all cases, further certificates may be required, for instance certificates for the keys of the authorities in question, and certificates that place `p`, `f`, `i`, and `o` in particular groups.

The client provides this material to its SCB, along with the data for the call. The SCB can then verify and assert `safe(f)`; it can similarly assert `may-read(p,i)` and `may-read(p,o)`. The client should present these signed assertions along with its call, and with a certificate that its SCB is in the group `GoodSCB`. Upon receipt of the call, the server automatically verifies that the SCB's assertions match its requirements before launching the execution of `f`.

The server can be even more forthcoming on its expectations. In particular, it can provide some information on how `safe(f)`, `may-read(p,i)`, `may-write(p,o)`, and `GoodSCB(s)` may be established. For instance, the server could supply a piece of code that implements `safe`, and a rule that expresses that if `z` is a good SCB and `v` is a good program then `z quoting v` may read `i` and write `o`:

```
public void compute(p : Principal,
                    f : Code,
                    i : FileName,
                    o : FileName)
requires for some r, s.
             p = (s quoting r),
             GoodSCB(s),
             s says safe(f),
             s says may-read(p,i),
             s says may-write(p,o),
where safe = (function x : Code.  /* checking */ ),
      for all q, v, z.
        q = (z quoting v) /\ GoodSCB(z) /\ GoodProgram(v)
```

```
          =>
        may-read(q,i) /\ may-write(q,o)
```

The client's SCB need not "believe" the definition of `safe` nor the rule for `may-read` and `may-write`. To the client's SCB, these are interpreted as assertions by the server, which may (or may not) be trusted on them.

In creating a concrete system to support examples such as this one, several design issues remain:

- exactly how to name the code for predicates, and how are names bound (e.g., early or late);
- how to sign certificates, and how to make assertions on secure channels;
- how freshness is specified and guaranteed;
- what is the syntax for describing interfaces (as annotations, perhaps in XML, or in an extension of an underlying programming language);
- what is the exact environment in which the code runs (e.g., in a virtual machine on top of the SCB, as suggested above).

Addressing these issues requires more careful engineering than difficult research.

9 Extensions

In this section we briefly consider variants and extensions of the ideas described above.

9.1 Checking with auxiliary state

A first, minor extension consists in taking into account auxiliary state that the SCB may keep. For instance, an SCB can certify network requests from its host up to some number (say, 1,000) per day. The requests may include calls on web services, such as search engines, and also requests to send e-mail (via SMTP) or to create free e-mail accounts. Of course, the requests can be broken into classes, with a different limit for each. Anyone that receives a non-certified request would have reason to suspect that it is generated by a program rather than a human user, and may disregard it or give it low priority. For this example, the SCB can simply rely on monotonic counters.

As this example shows, one advantage of performing checks at the caller's SCB is that the SCB can rely on any relevant auxiliary state it can keep. The state may not readily be available at the callee.

9.2 Supplying inputs

In further extensions, an SCB may do more than checking data: it can supply all or part of an input. The SCB can thus guarantee that the input is generated in a certain way. For example, the SCB can guarantee that the input is generated

- with a particular protocol stack,

- by running a particular compiler,
- with inlined safeguards that enforce a security policy, such as an inlined reference monitor [12],
- with a particular application (for example, with a trusted tax-preparation package),
- by completing a particular form, or
- directly by a user, through secure I/O.

Although these scenarios may be attractive, some of them may require running substantial pieces of code on the SCB. These scenarios often tend to fit into a fairly controlled approach to systems, which enforces not only what hosts say but also why they say it (what code they run).

In addition, the SCB can help when the input in question contains sensitive information (such as personal medical records). The SCB may be in charge of holding the sensitive information, and occasionally encrypting it and sending it to designated parties, or displaying it on a trusted output device. In such examples, the SCB is involved not in order to guarantee how the data is generated, but in order to protect its secrecy.

9.3 General mechanisms, revisited

The general mechanisms described in section 7 can extend to the cases in which the SCB maintains auxiliary state and in which it supplies part or all of an input. However, some problems arise. For instance, the servers would need means of naming auxiliary data at the clients's SCBs. In addition, the security policies that protect clients from servers may become more elaborate. While these problems do not appear difficult, it may be premature to invest in their solution.

10 Conclusions

Trusted Computing gives rise to a new supply of potential trusted third parties. These trusted third parties may find a variety of applications in distributed systems—keeping sensitive personal information, preventing cheating in games, and possibly many more.

In this paper we investigate the use of these trusted third parties for verified communications. We consider several instances of remote input checking, such as remote typechecking, proof checking, and certificate checking. We also outline a general mechanism for these purposes.

Despite the lively controversy on Trusted Computing, and despite the substantial progress in the development of its basic machinery, there remains much room for further thinking and experimentation. In particular, this thinking and experimentation should shed more light on the potential uses of this technology, which are important whether one prefers Trusted, Trustworthy, or Treacherous Computing.

Acknowledgements

This paper was written at Microsoft Research, Silicon Valley, and is partly based on discussions with Andrew Birrell, Mike Burrows, Luca Cardelli, John DeTreville, Cynthia Dwork, Úlfar Erlingsson, Cédric Fournet, Andy Gordon, Jim Gray, Butler Lampson, Paul Leach, Roy Levin, Greg Morrisett, Chuck Thacker, Chandu Thekkath, and Ted Wobber. Chandu Thekkath and Ted Wobber also suggested improvements to a draft of this paper. Thanks to all of them.

References

[1] Ross Anderson. Cryptography and competition policy - issues with 'Trusted Computing'. On the Web at `www.ftp.cl.cam.ac.uk/ftp/users/rja14/tcpa.pdf`, 2003.

[2] Ross Anderson. 'Trusted Computing' Frequently Asked Questions - TC / TCG / LaGrande / NGSCB / Longhorn / Palladium / TCPA. Version 1.1, on the Web at `www.cl.cam.ac.uk/~rja14/tcpa-faq.html`, August 2003.

[3] Ross Anderson and Roger Needham. Robustness principles for public key protocols. In *Advances in Cryptology—CRYPTO '95*, pages 236–247. Springer, 1995.

[4] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 52–62, November 1999.

[5] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the Web. In *Proceedings of the 11th USENIX Security Symposium 2002*, pages 93–108, 2002.

[6] Don Box, Chris Sells, and Ted Pattison. *Essential .NET Volume I: The Common Language Runtime*. Addison Wesley, 2002.

[7] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[8] Robert DeLine and Manuel Fähndrich. Enforcing High-Level protocols in Low-Level software. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 59–69, June 2001.

[9] John DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 105–113, May 2002.

[10] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology—CRYPTO '92*, pages 139–147. Springer, 1992.

[11] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *IEEE Computer*, 36(7):55–62, 2003.

[12] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255. IEEE Computer Society Press, 2000.

[13] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP 2003)*, pages 193–206, October 2003.

[14] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, pages 526–538. Springer, 2002.

[15] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, 2nd edition, 2003.

[16] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Proceedings of the IFIP TC6 and TC11 Joint Working Conference on Communications and Multimedia Security (CMS '99)*, pages 258–272. Kluwer, 1999.

[17] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[18] Xavier Leroy. Bytecode verification on Java smart cards. *Software — Practice and Experience*, 32(4):319–340, April 2002.

[19] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Ninth International ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[20] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.

[21] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[22] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[23] George C. Necula. A scalable architecture for proof-carrying code. In *Functional and Logic Programming, 5th International Symposium, FLOPS 2001*, pages 21–39. Springer, 2001.

[24] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP 1998)*, volume 1384, pages 235–246. Springer, 1998.

[25] Richard Stallman. Can you trust your computer? On the Web at `www.gnu.org/philosophy/can-you-trust.html`, 2002.

[26] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.