Approved for Public Release; Distribution Unlimited
Case # 10-0959

© The MITRE Corporation. All rights reserved

Attestation Turns Crash Tolerance into Byzantine Tolerance

Jonathan Herzog *
MIT Lincoln Laboratory
jherzog@ll.mit.edu
* *Work performed prior to employment by MIT Lincoln Laboratory*

Jonathan Millen, Brian O'Hanlon, John D. Ramsdell, and Ariel Segall
The MITRE Corporation
jmillen,bohanlon,ramsdell,asegall@mitre.org

***Abstract*—An attestation protocol enables one node in a distributed system to detect nodes that are Byzantine due to a malicious intrusion. Using attestation, distributed algorithms that tolerate channel crash failures can be transformed into ones that also tolerate Byzantine failures. The idea is to provide a network interface that requires a successful attestation before permitting messages from a remote nodes to be received. Thus, channels to Byzantine nodes are made to appear crashed. Erlang modules to support filtering and attestation have been written, including a partial Trusted Platform Module (TPM) interface.**

I. Introduction

Malicious intrusions in nodes of distributed systems can be viewed as Byzantine faults. With various assumptions about network behavior and attacker capability, distributed algorithms can be written that tolerate some proportion of Byzantine failures. However, the available solutions are better when it is necessary to deal only with crash failures. If the only possible failures are crash failures, and nodes can detect crashes (though timeouts, for example) then distributed agreement can be achieved so long as at least two participants remain alive [11]. But even under the most optimistic of settings—completely synchronous processes and reliable channels—agreement is not possible if less than two-thirds of the processes are honest (non-Byzantine).

Furthermore, if an agreement algorithm is resilient against Byzantine failures, then it must be time-consuming. The synchronous algorithms for Byzantine agreement proceed in rounds, and such protocols must use at least one round more than the number of Byzantine faults. The asynchronous agreement algorithms generally emulate the synchronous algorithms and proceed in rounds, but the need to simulate synchronicity in an asynchronous setting generally increases the cost of each round.

The core of the problem is the inability to test for Byzantine faults, so that Byzantine nodes must be outvoted without knowing which ones they are.

A. TPMs and Attestation

By taking advantage of some recent technological advances, intrusions or faults in a system that permit Byzantine behavior can, in many cases, be detected and reported by protected subsystems.

A *protected subsystem* is a service or component on a node that is, to an adequate degree of assurance, free from bugs and malicious intrusions. It can act as a root of trust for measuring and reporting on the health of the rest of the node. It can be destroyed but it cannot be made Byzantine or successfully emulated. Various mechanisms, usually with hardware support or dependencies, have been suggested for implementing a protected subsystem capable of initiating and saving suitable local measurements. An *attestation protocol* is a way for a remote node to request and obtain a measurement report from the protected subsystem of a target node to be evaluated.

In a setting proposed by the Trusted Computing Group (TCG), each computing platform would have a Trusted Platform Module (TPM)—a hardware module that can store the 'fingerprint' of a platform's boot-up sequence [3]. The TPM can also sign these fingerprints with a unique key, certified as belonging to a TPM, so that TPM reports cannot be falsified. Computers manufactured by members of the TCG already have these installed. A TPM, combined with software support, can produce a platform capable of attestation.

Attestation protocols can also invoke and report run-time measurements made by tools such as LKIM for Linux kernel inspection [10]. By invoking such measurements, a remote node can detect intrusions into system software that have occurred subsequent to boot-up.

In a system with a virtual machine monitor or hypervisor, an attestation protocol can report on an individual virtual machine (VM) by combining the TPM report on the hypervisor with a measurement of the kernel or other software within a particular VM. Thus, attestation can deal with distributed systems in which several logical nodes are actually VMs co-located on the same physical platform. A discussion of how attestation would work in such a system is given in [6].

At a high level, our concept is independent of the details of how attestation is supported, but our prototype implementation work is designed to take advantage of TPM services, and it is compatible with future support for kernel measurements.

© The MITRE Corporation. All rights reserved

In short, attestation technology may make it possible for one node to determine whether another node is actually Byzantine. It should be admitted that, at present, no measurement and attestation technology is guaranteed to identify exactly the Byzantine nodes. Even now, however, attestation methods offer considerable improvement over their absence. For now, in the theoretical result, we look at the consequences of the simplifying assumption that Byzantine failures are always detectable by attestation. Even if attestation does not detect all Byzantine nodes, however, it can at least reduce the number of undetected ones, allowing some systems to conform to a failure threshold that would not otherwise be satisfied.

B. Adding a Dispatcher to a Distributed System

With our approach, illustrated in Fig. 1, each node has a user process running its role in the distributed algorithm. In addition, there is a local 'dispatcher' process on the same platform acting as a proxy for all network communication. On the first attempt at a message exchange with another node, the dispatcher initiates an attestation protocol so that the dispatcher of the receiving node can determine whether the sending node is Byzantine. If not, the attestation protocol generates and shares a session key for encrypting messages in this direction. If so, the receiving dispatcher will refuse to provide a session key. No messages will be received unless they have been encrypted with a proper session key, so an attestation failure will block messages from a Byzantine node. The channel from that node therefore appears to have undergone a crash failure (which may or may not be distinguishable from a crash failure of the remote node itself).

This means that the benefits of the dispatcher are independent of the extent of crash tolerance offered by the original distributed algorithm. The dispatcher simply protects the honest nodes from Byzantine attacks, but not from crashes.

When the dispatcher engages in an attestation protocol, there are actually three possible results: (1) the attestation is a success and communication with the remote node is enabled; (2) the attestation fails and the dispatcher blocks messages from the remote node; or (3) the attestation protocol does not terminate, either because of communication delay or because of noncooperation from the remote node. Inability to receive a message due to nonresponse from the sending dispatcher just looks like an asynchronous network communication delay, which the distributed algorithm is expected to handle somehow.

Note that a remote system may be compromised in such a way that its own local dispatcher is also compromised. Attestation protocols are designed so that the challenging dispatcher will not be fooled by a malicious remote dispatcher. The ability to guarantee this rests on the essential root of trust in the protected subsystem.

C. Failure Detector Support

Some distributed algorithms are designed to make use of a *failure detector*[5], [4]. A failure detector for crashes is a function that can explicitly ”suspect” a remote node as having crashed, so that the rest of the algorithm can take appropriate action without waiting indefinitely. This is sometimes implemented by imposing a time-out. There are several possible models for the properties of a failure detector. In particular, their suspicions are not required to be correct, although they may be expected to converge on correctness if invoked repeatedly on the same target.

From the dispatcher's point of view, a failure detector is part of the algorithm, and its presence does not affect the attestation. However, a dispatcher can support a failure detector by passing along the negative result when it gets one, so that the failure detector can report the dispatcher-induced crash to the user process.

D. Implementation Support in Erlang

To demonstrate the dispatcher architecture, we chose Erlang as an implementation language. [2], [1] This language, designed for the implementation of distributed systems and algorithms, natively supports such high-level features such as crash-failure detection and reliable channels. Erlang is a parallel functional programming language designed for programming real-time control systems such as telephone exchanges and automated teller systems. The compiler and runtime environment are available under an open source license and as downloadable object code. For any implementation, we have to consider (1) how to cause the dispatcher to be invoked, (2) how the dispatcher authenticates and filters external network communication, (3) how to design, verify, and implement an attestation protocol, and (4) how to make use of a TPM, if we depend on it to record and report measurements.

We implement four things in Erlang:

- A *parse transform* that can be applied to Erlang source code, forcing the dispatcher to be invoked;
- Our dispatcher, which employs an attestation protocol;
- An attestation protocol, which uses a TPM interface to obtain stored boot measurements;
- An Erlang interface to access a TPM.

The TPM interface requires an intermediate interface in C which will also be described briefly.

II. Modeling Approach

We can show that when attestation is perfect, Byzantine fault tolerance is actually *no harder* than tolerance against channel failures. The result is only sketched here; the details are in [9]. Our modeling style is based on that of Lynch [11], using I/O-Automata (IOA). An IOA model has concurrent processes that communicate instantaneously by sharing events that are outputs of one process and inputs to any other process capable of receiving them.

© The MITRE Corporation. All rights reserved

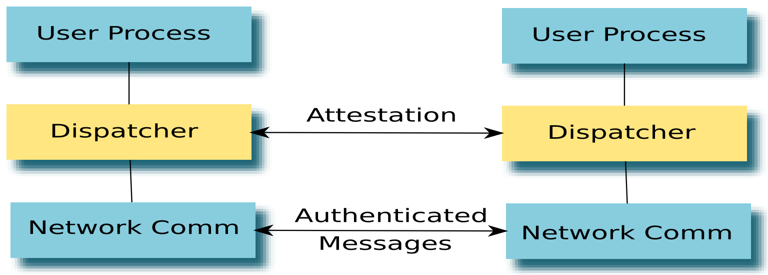

Figure 1. Dispatcher Architecture

Asynchronous network delays are modeled by adding *channel* processes that maintain message queues between pairs of communicating processes, so that sending a message is a separate event from receiving that message. This model assumes that order is preserved among messages from one node to another. The model also includes an "anonymizer" process that collects inputs to a given node from all the channels it receives from. The anonymizer merges the messages it receives into a single queue, thus losing source information, before passing them on to the user process. This makes it possible to model attempts by Byzantine processes to falsify the source of a message.

There are three steps to the result:

- First, we define an 'ideal' network model with the components just described, in which crash faults are possible but Byzantine faults are not. That is, this model allows nodes to crash and recover, channels to crash and recover, and individual messages can be dropped. These events are caused by external inputs to nodes and channels, representing spontaneous faults.
- We then define a 'real' network model that contains all of the failures of the 'ideal' model but with node Byzantine failures as well. More details of the 'real' network model are given below.
- We then show trace equivalence between the two network models. That is, we show that every distributed algorithm induces exactly the same set of possible traces in both network models. A trace is defined to include only "visible" events, which are those incident to honest user processes.

This means that the real network with dispatchers is behaviorally indistinguishable to the honest nodes from a network with possible channel crash failures but no Byzantine failures. Thus, in settings where attestation can be performed, channel-crash-resilient distributed algorithms can automatically become Byzantine-resilient as well.

A. *The 'Real' Network Model*

In the 'real' network model, some user process nodes are identified as Byzantine. Byzantine nodes do not fail and recover; they are permanently Byzantine, at least over the time scale in which the distributed algorithm operates. The real network has some additional processes beyond the ideal one. Each node is augmented with a *dispatcher* process, which acts as a network proxy. Dispatchers use an *attester* process to determine which other nodes are honest and which are Byzantine, so that the dispatcher can prevent messages from Byzantine processes from reaching honest nodes. Each dispatcher has state information recording the known status of other nodes.

In the model, the behavior of a Byzantine user process, and of its dispatcher, turns out to be irrelevant; no assumptions are made about it. The important assumption is that the dispatcher of an honest process, using an appropriate attester, never makes an incorrect assessment of a remote node. The model has enough detail to represent the use of shared session keys in each direction between honest user processes, thereby detecting messages from Byzantine processes. The 'real' network model also provides for an 'apparent sender' of a message so that a Byzantine process can attempt to fool a dispatcher with messages that appear to come from an honest process. Such attempts are shown to be defeated.

B. *Comparison With Other Approaches*

There has been some prior work on Byzantine failure detection applicable to agreement algorithms. A paper by Doudou, Garbinato, and Guerraoui [7] proposes a protocol that detects a limited type of Byzantine misbehavior, called a *muteness failure*, in which messages to a chosen recipient may be deliberately stopped. Malkhi and Reiter [12] previously studied "quiet" processes, defined similarly, and proposed a protocol based on the assumption of an abstractly defined failure detector for them.

The work of Han, Pei, Ravindran, and Jensen [8] provides Byzantine tolerance for information dissemination with a gossip-based, real-time protocol. Gossip protocols disperse information gradually by sending messages to randomly selected group members, and their resiliency properties are probabilistic. The gossip-based protocol context makes it possible to define and detect Byzantine failures behaviorally.

Another example of an approach that deals with Byzantine failure in the context of a particular algorithm is [14].

© The MITRE Corporation. All rights reserved

They give an agreement and leader election algorithm for a network in which channels may have Byzantine faults.

Our work differs from prior approaches in that it can address whatever form of Byzantine fault is detectable by remote attestation, and it can be applied to any higher-layer distributed algorithm. The failure tolerance offered by the result depends on the crash tolerance of the higher-layer algorithm. And if some Byzantine faults slip through, and the higher-layer algorithm must deal with them, there are fewer to deal with.

III. The Components of the Erlang Implementation

A. The Parse Transform

Erlang provides simple, high-level primitives for message-passing and failure-detection as part of the language's core functionality. While the use of these primitives (and of Erlang in general) make Erlang programs clear and easy to understand, it 'hard-codes' the implementation into the the direct-communication model.

For example, consider the Erlang expression for sending a message with content `V` to the process (either local or remote) with ID `PID`:

`erlang:send(PID,V)` (or simply `PID ! V`)

In the dispatcher model, all communication must be re-routed through the recipient's dispatcher. Erlang provides no way for the dispatcher to automatically 'intercept' messages sent to its client, so every send expression in the code must be rewritten. The Erlang compiler provides functionality for rewriting code by using a *parse transform*, an Erlang module which provides functions to map Erlang parse-trees to Erlang parse-trees. The parse tree which is output is then compiled normally. As a result, we can automate morphing Erlang code into code that uses a dispatcher.

The parse transform does as little rewriting as possible. For the most part, it translates calls to functions in the `erlang` module which potentially involve other nodes to calls to modified functions in the `dispatcher` module. For example, the message-send syntax becomes

`dispatcher:send(PID,V).`

The functions in the `dispatcher` module produce the same effect as their built-in forms, except that Byzantine nodes will be made to look crashed.

The rest of the translation involves message receipt. Messages should only be recieved directly from the dispatcher, and only be of a certain form. The simplest form of a receive statement in Erlang is:

```
receive
  Pattern -> Body
end
```

where the `Pattern` specifies a structure or schema for an acceptable message, and the `Body` has the code executed when such a message is received.

The transformed statement is:

```
NONCE = dispatcher:get_nonce(),
receive
 {from_dispatcher,NONCE,Pattern} -> Body
end
```

The new pattern contains a fixed constant `from_dispatcher` as the first component, a nonce (large random number) as the second component, and the original message as the third component. A message will match the new pattern if and only if the original message would match, and the nonce authenticates that the message came from the dispatcher. The second component provides the security – the nonce is matched against a previously established secret value shared between the dispatcher and the node. This mechanism is weak, but we are also constrained by the Erlang language, which does not allow side-effects which may be caused by more sophisticated cryptographic mechanisms in guard sequences.

B. The Dispatcher

When the code is modified with the parse transform during compilation, calls to message sending and receiving commands, as well as other node maintenance commands in the Erlang standard library, become calls to functions in our dispatcher.

The dispatcher is implemented as a *named process* which, if it is not already running, is launched at each user node upon receipt or sending of a message. Thus, `dispatcher:send` actually sends the local dispatcher process a message. The dispatcher maintains state about which node keys it knows, and it loops to handle service request messages and incoming messages.

When the dispatcher is requested to send or receive a message to or from a remote node, the first thing it does is check if an attestation needs to be initiated. If so, the attestation protocol described in Section refsec:protocol must be invoked. If the message is to be sent, a successful attestation results in an `outkey`, to be used on any outgoing message to that remote node. If the request is to receive a message, the attestation should result in an `inkey`, which is applied to any incoming message from that remote node. If an `inkey` is not present, incoming messages are dropped.

C. EVA: A Simple Attestation Protocol

For our prototype, we created an attestation protocol called EVA, inspired by the more complex CAVES protocol [13]. Only two parties participate in EVA: an Attester, which initiates the attestation protocol, receives attestation requests and interacts with the TPM, and a Verifier, which determines whether to issue a shared key to the Attester. These are roles handled by the dispatchers at the responder and the requester end, respectively. The attestation protocol

© The MITRE Corporation. All rights reserved

is oriented so that the dispatcher of the node that wishes to send a message runs the Attester role and the one that wishes to receive it runs the Verifier role.

We assume for this protocol that the parties are known to each other, and share any public keys necessary for confirming the identity of other participants. Each party has a unique name, a public key associated with it, and TPM identity key associated with the same user name. These associations may be pre-loaded or established through an exchange of public-key certificates. At the end of the protocol, if the Verifier is satisfied with the reported measurement, the Verifier and Attester should share a session key.

Our security goals for this protocol were that the Verifier should know that the report is fresh, it comes from a legitimate TPM on the platform where the Attester is running, and that the shared session key k_s is known only to the two legitimate participants.

$$\begin{tikzcd}
\text{Attester} \arrow[rr, "(\texttt{connect\_request}{,}a{,}v)"] \arrow[d, Rightarrow] & & \text{Verifier} \arrow[d, Rightarrow] \\
\bullet \arrow[d, Rightarrow] & & \bullet \arrow[ll, "\{\!|n_v{,}v{,}m|\!\}_A"'] \arrow[d, Rightarrow] \\
\bullet \arrow[rr, "\{\!|k_s{,}v{,}m{,}p{,}\{\!|\texttt{hash}(\texttt{hash}(a{,}v{,}n_v){,}m{,}p)|\!\}_{I^{-1}}|\!\}_V"] \arrow[d, Rightarrow] & & \bullet \arrow[d, Rightarrow] \\
\bullet & & \bullet \arrow[ll, "\{\!|\texttt{approved}{,}k_s|\!\}_A"']
\end{tikzcd}$$

In this protocol, a is the name of the Attester and v the name of the Verifier. Their public keys are A and V, respectively. The Attester's TPM identity key is I. Encryption of x by k is symbolized by $\{\!|x|\!\}_k$, and digital signature is represented as an encryption with the inverse key. Freshness is guaranteed through the use of a nonce n_v generated by the Verifier. The session key k_s is generated by the Attester. The report from the TPM is a vector p of Platform Configuration Register (PCR) values; the particular registers requested are specified in the mask m. The hash of a, v, n_v is passed from the attester process to the TPM, which binds the PCR values to it in the signed report. The Verifier is assumed to recognize the binding of a to I through prior exchange of a certificate, and it will approve the measurements p before confirming the session key. The expression signed by the TPM key, incidentally, is called a "TPM Quote".

We have verified this protocol using the tool CPSA [13] and demonstrated that all of our desired security properties have been met. One of the objectives of the formal verification is to check that the protocol is not subject to network attacks, such as a man-in-the-middle attack, which might allow a Byzantine node to "borrow" a good measurement report from another, honest node.

D. Interface to the Trusted Platform Module

The standard mechanism for using the Trusted Platform Module (TPM) today is to go through one of the libraries implementing the TCG Software Stack (TSS) interface[16]. We used the open-source Linux implementation, TrouSerS. In order to access the TPM's functionality in our distributed Erlang code, we needed two additional levels of interface:

- an Erlang TPM server process with functions required by the attestation protocol, and
- a C node accessible via an Erlang *port*.

The TPM server process accepts TPM command request messages from any dispatcher on the platform; it is essentially a proxy for the TPM. The server is implemented as an Erlang script that translates Erlang messages and data structures into bit-string inputs for the C node, which makes direct calls on TrouSerS.

The Erlang interface provides access to the following functions essential to remote attestation:

load: Takes a TPM identity key pair (in encrypted "blob" form) and loads the secret key into the TPM for use in subsequent quote commands.

quote: Takes a bit mask specifying PCR choices and a user-supplied value referred to as a "nonce", and causes the TPM to return a TPM Quote with an identity-key signature. The quote includes a composite hash of the requested PCR values and the nonce.

verify: Takes a public identity key and a quote result (both from a remote node) that can be validated by that identity key, and returns the composite hash of the PCR contents and the nonce encapsulated in the quote, provided that the signature is valid.

There is additional supporting function in a separate program for creating an identity key through TSS.

IV. Ongoing Activities

For purposes of demonstration, we are implementing a prototype distributed system that exercises the dispatcher architecture as described. To do so, we had to select a crash-tolerant algorithm as an example. Our initial demonstration used a leader-election algorithm that was already implemented in Erlang and distributed with the Erlang package [15]. However, we plan to move to a different algorithm in order to coordinate with other research projects at our company. Our objective is to implement an algorithm that provides Byzantine tolerance, in order to show how it benefits from a reduction in the number of undetected Byzantine nodes. We also want to design and analyze a dispatcher that does not assume that a successful attestation is permanent, so that subsequent re-attestation is performed according to a suitable risk-reduction policy.

We have found that the C interface to the TCG software has engendered interest from other projects that found the full TSS interface difficult to use. We intend to create demonstrations for other distributed algorithms and also to find ways to implement the dispatcher concept in other languages and architectures, and with additions and alternatives

© The MITRE Corporation. All rights reserved

for the kinds of measurements and protected subsystems employed.

Acknowledgements

The authors are grateful for our stimulating discussions with Doug Jensen and Leonard Monk, both of whom helped us adjust our perspective.

References

[1] Joe Armstrong. *Making reliable dstributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, November 2003.

[2] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Programmers, 2007.

[3] Boris Balacheff, Liqun Chen, Siani Pearson (ed.), David Plaquin, and Graeme Proudler. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall PTR, Upper Saddle River, NJ, 2003.

[4] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[6] George Coker, Joshua D. Guttman, Peter Loscocco, Justin Sheehy, and Brian T. Sniffen. Attestation: Evidence and trust. In Liqun Chen, Mark Dermot Ryan, and Guilin Wang, editors, *Proceedings, the 10th International Conference on Information and Communications Security (ICICS)*, volume 5308 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[7] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: from crash to Byzantine failures. In *Ada-Europe 2002*, volume 2361 of *LNCS*, pages 24–50. Springer, 2002.

[8] Kai Han, Guanhong Pei, Binoy Ravindran, and E. Douglas Jensen. Real-time, byzantine-tolerant information dissemination in unreliable and untrustworthy distributed systems. In *ICC*, pages 1727–1731, 2008.

[9] Jon Herzog, Jon Millen, Brian O'Hanlon, John Ramsdell, and Ariel Segall. Using attestation to lift crash resilience to byzantine resilience. Technical report, The MITRE Corporation, 2009.

[10] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux kernel integrity measurement using contextual inspection. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable Trusted Computing*, pages 21–29, New York, NY, USA, 2007. ACM.

[11] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[12] D. Malkhi and M. Reiter. Unreliable intrusion detection in ditributed computations. In *10th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1997.

[13] John D. Ramsdell, Joshua Guttman, Jonathan Millen, and Brian O'Hanlon. An analysis of the CAVES attestation protocol using CPSA. November 2009. MTR090213.

[14] H. Sayeed, M. Abu-Amara, and H. Abu-Amara. Optimal asynchronous agreement and leader election algorithm for complete networks with byzantine faulty links. *Distributed Computing*, 9(3):147–156, 1995.

[15] Hans Svensson and Thomas Arts. A new leader election implementation. In *Proceedings of the ACM SIGPLAN 2005 Erlang Workshop*, pages 35 – 39, September 2005.

[16] Trusted Computing Group. *TCG Software Stack (TSS) Specification*, version 1.2 level 1 errata a edition, 2007. http://www.trustedcomputinggroup.org/developers/software_stack.