PRINT CLOSE

(*IEEE SECURITY & PRIVACY* EXCLUSIVE CONTENT)
ON THE HORIZON

Intrusion-Tolerant Middleware: The Road to Automatic Security

Paulo E. Veríssimo, Nuno F. Neves, Christian Cachin, Jonathan Poritz, David Powell and Yves Deswarte, Robert Stroud, and Ian Welch

JUL./AUG. 2006

The pervasive interconnection of systems throughout the world has given computer services a significant socioeconomic value that both accidental faults and malicious activity can affect. The classical approach to security has mostly consisted of trying to prevent bad things from happening—by developing systems without vulnerabilities, for example, or by detecting attacks and intrusions and deploying ad hoc countermeasures before any part of the system is damaged. But what if we could address both faults and attacks in a seamless manner, through a common approach to security and dependability? This is the proposal of *intrusion tolerance*, which assumes that

- systems remain somewhat faulty or vulnerable;
- attacks on components will sometimes be successful; and
- automatic mechanisms ensure that the overall system nevertheless remains secure and operational.

No large-scale computer network can be completely protected from attacks or intrusions. Just as chains break at their weakest link, any inconspicuous vulnerability left behind by firewall protection or any subtle attack that goes unnoticed by intrusion detection will be enough to let a hacker defeat a seemingly powerful defense. Using ideas from fault tolerance that put emphasis on automatically detecting, containing, and recovering from attacks, the European project MAFTIA (Malicious-and Accidental-Fault Tolerance for Internet Applications) set out to develop an architecture and a comprehensive set of mechanisms and protocols for tolerating both accidental faults and malicious attacks in complex systems. Here, we report some of the advances made by the several teams involved in this project, which brought together international expertise in the areas of information security and fault tolerance.

Intrusion tolerance in a nutshell

Building an intrusion-tolerant system to arrive at some notion of intrusion-tolerant middleware for application support presents multiple challenges. Surprising as it might seem, intrusion tolerance isn't just another instantiation of accidental fault tolerance.

To capture the essence of intrusion tolerance, we must first consider that an intrusion is in fact a malicious fault that has two underlying causes: a weakness, flaw, or *vulnerability*, or a malicious act or *attack* that attempts to exploit the former.

Attacks, vulnerabilities, and intrusions

Vulnerabilities are the primitive faults within a system—in particular, design or configuration faults—that can be introduced during the system's development or operation. For example, as a step in an overall plan of attack, an attacker might introduce vulnerabilities in the form of malware.

Attacks are malicious faults that attempt to exploit one or more vulnerabilities. An attack that successfully exploits a vulnerability results in an intrusion, which is normally characterized by an erroneous system state (for example, a system file with unwarranted access permissions for the attacker). If nothing is done to handle these errors, a security failure will probably occur. Attacks often assume the form of inconsistent interactions with different legitimate participants in order to confuse them. Resilient systems should be able to handle these so-called *Byzantine faults*.

Figure 1a represents a fundamental sequence: attack → vulnerability → intrusion → error → failure. This well-defined relationship is called the *AVI fault model*.

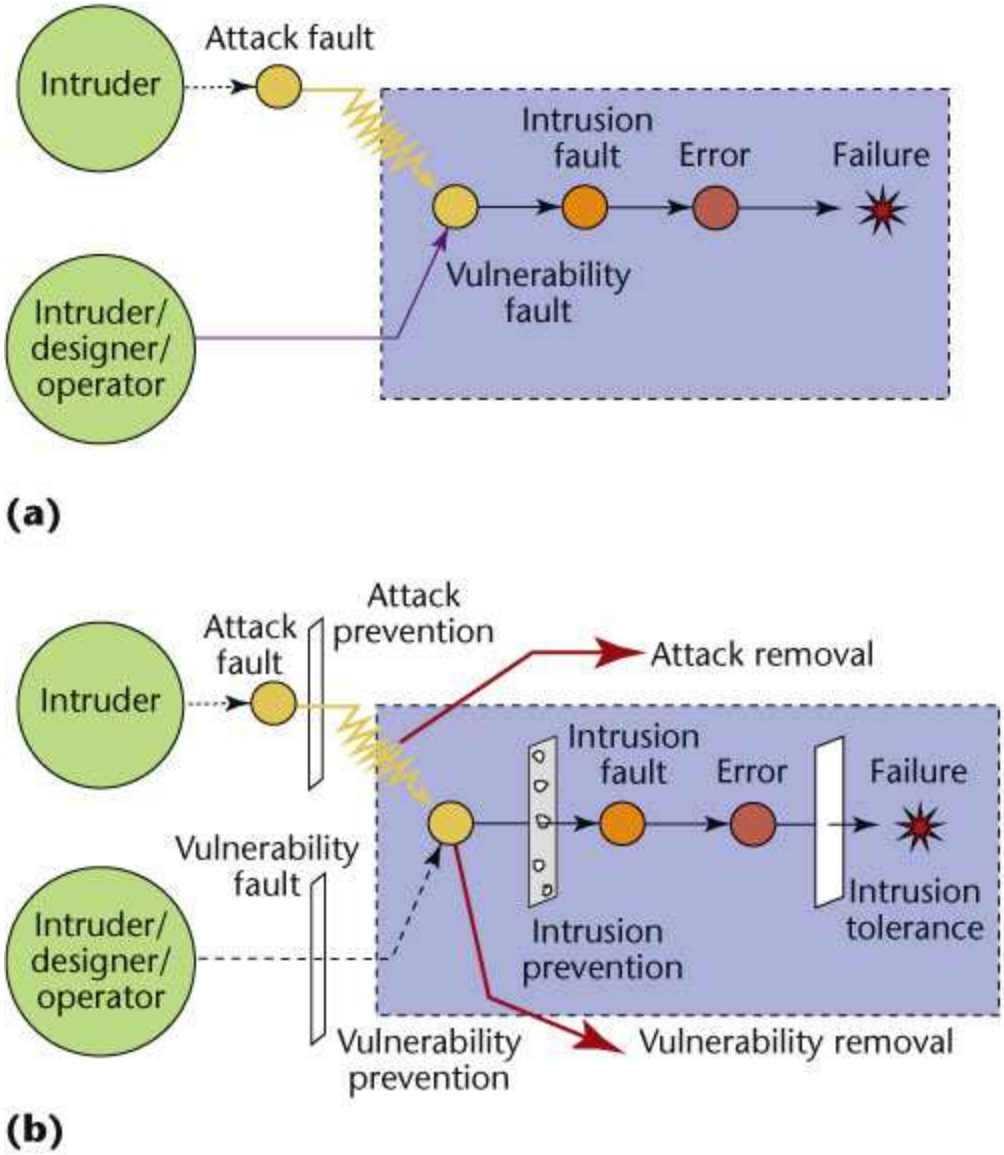

Figure 1. Intrusion sequence. In the (a) attack-vulnerability-intrusion (AVI) fault model, an attack hits a vulnerability, causing an intrusion which, if not handled, will cause a failure; (b) intrusion tolerance, the last resort for protection.

Classical security methodologies mainly focus—quite successfully—on preventing intrusion. However, as reality painfully proves every day, it's impossible, even infeasible, to guarantee perfect prevention: simply put, we can't handle all attacks because they aren't all known, and new ones appear constantly. As a consequence, a few inconspicuous weaknesses are easy prey to hackers, and what's worse, the resulting intrusions that escape the *intrusion-prevention* barrier, as Figure 1b suggests, will go unnoticed and will likely cause security failures.

The last resort is intrusion tolerance, which, as the name suggests, acts after intrusion and before failure. Intrusion-tolerance techniques are in essence automatic, relying on local mechanisms and distributed protocols, and assume combinations of detection (of corrupted hosts or tampered communications), recovery (neutralization of intruder activity), or masking (use of spare components or replicas, such that the whole resists the intrusion of a minority).

Trust and trustworthiness

The relationship between the notions of *trust* and *trustworthiness* is important to understand how intrusion tolerance can lead to secure designs.

Let's consider trust to be a component's accepted dependence on a set of (desirable) properties of another component,

subsystem, or system.[1] If *A* trusts *B*, then *A* accepts that a violation of *B*'s properties might compromise A's correct operation. It might also happen that those properties *A* trusts don't correspond quantitatively or qualitatively to *B*'s actual properties. Thus trustworthiness is the measure in which a component (say, *B*) meets a set of properties. Clearly, a robust design implies that trust in *B* should be placed to the extent of *B*'s trustworthiness—that is, the relation "*A* trusts *B*" should imply *A*'s substantiated belief that *B* is trustworthy in the measure of *B*'s trustworthiness.

There is a separation of concerns between how to make a component trustworthy (constructing the component) and what to do with the trust placed in the component (building fault-tolerant algorithms). These iterative chains of trust–trustworthiness relations, with the proper specification and verification tools—lead to very clear arguments about system security and dependability.[2,3]

MAFTIA architecture

The MAFTIA architecture selectively uses intrusion-tolerance mechanisms to build layers of progressively more trusted components and middleware subsystems from baseline untrusted components (hosts and networks). This leads to an automation of the process of building resilience: at lower layers, a trustworthy communication subsystem is constructed with basic intrusion-tolerance mechanisms. Higher-layer distributed software can then trust this subsystem for secure communication among participants without worrying about network intrusion threats. Alternatively, an even more trustworthy higher layer can be built on top of the communication subsystem—by incrementally using intrusion-tolerance mechanisms—such as a replication management protocol that's resilient against both network and host intrusions.

Architectural options

A MAFTIA host's structure relies on a few main architectural options, some of which are natural consequences of the discussions in the previous section:

- The notion of trusted—versus untrusted—hardware. Most of MAFTIA's hardware is untrusted, but small parts of it are trusted to the extent of some quantifiable measure of trustworthiness—for example, being tamper-proof by construction.
- The notion of trusted support software that can execute a few functions correctly, albeit in an environment subjected to malicious faults.
- The notion of a runtime environment that extends operating system capabilities and hides heterogeneity among host operating systems by offering a homogeneous API and framework.
- The notion of trusted distributed components, materialized by MAFTIA middleware, which are modular and multilayered. Each layer overcomes lower layers' faulty behavior.

We can depict the MAFTIA architecture in at least three different dimensions (see Figure 2). The *hardware* dimension includes the host and networking devices that compose the physical distributed system. Within each node, the operating system and runtime platform (which can vary from host to host) provide *local support* services. Finally, MAFTIA provides *distributed software*: the layers of middleware running on top of the runtime support both the mechanisms that each host provides and MAFTIA's native services—authorization, intrusion detection, and trusted third parties. To operate securely across several hosts even in the presence of malicious faults, applications running on MAFTIA use the abstractions that the middleware and application services provide.

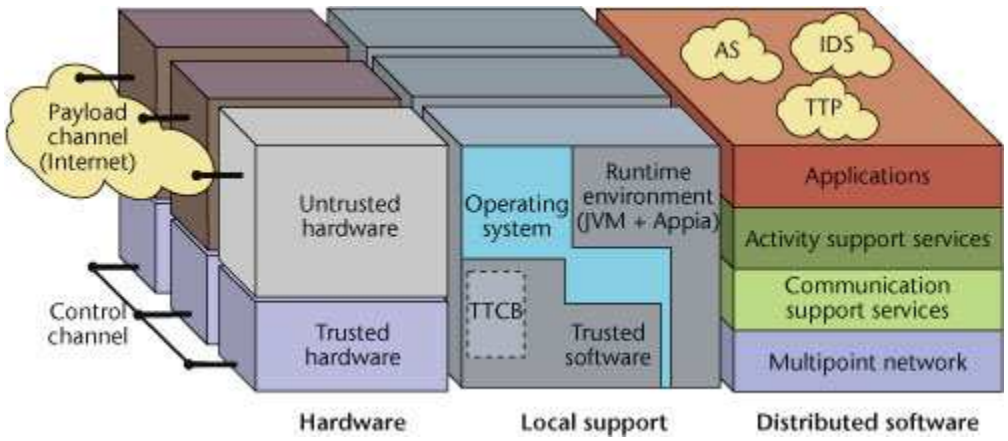

Figure 2. The MAFTIA architecture's three dimensions. Hardware, local support, and distributed software help applications operate securely across several hosts, even in the presence of malicious faults.

Hardware

We assume that the hardware in individual MAFTIA hosts is untrusted in general. In fact, most of a host's operations run on untrusted hardware—such as the usual PC or workstation machinery connected through the normal networking infrastructure to the Internet, which we call the *payload channel*. However, some hosts might have pieces of hardware that are trusted to the extent of seeming tamper-proof (that is, we assume intruders don't have direct access to the inside of these components). MAFTIA features two incarnations of such hardware, both of which are easy to incorporate in standard machines because they're commercial-off-the-shelf (COTS) products. One is a *smart card* (actually a Java card), connected to the machine's hardware and interfaced by the operating system. The other is an *appliance board*,which has a processor and an adapter to a (trusted) control network. The runtime support has specialized functions provided by the trusted support software and implemented in two components, the Java Card Module (JCM) and the Trusted Timely Computing Base (TTCB). For less demanding configurations, we also designed a software-implemented TTCB.[4]

Local runtime support

The MAFTIA architecture's runtime support dimension essentially consists of the operating system augmented with appropriate extensions. The middleware, service, and application software modules run on the Java virtual machine (JVM) runtime environment. The JCM assists the operation of a reference monitor that supports the MAFTIA authorization service.[5] This reference monitor checks all accesses to local objects and autonomously manages all access rights for local objects. We trust the Java card to be tamper-proof for application services whose value is much less than the effort—in means or time—necessary to subvert it.

Distributed runtime support

The TTCB is a distributed trusted component responsible for providing a basic set of trusted time and security services to middleware protocols for communication and activity support. The TTCB services are accessed locally through runtime support but can have global reach, such as making a value known to all local TTCB parts, thus limiting the potential for Byzantine faults by malicious protocol participants, as we discuss later. We can assume that the TTCB component is infeasible to subvert, but it might be possible to interfere with its software component interactions through the JVM. Although this exposes a local host to compromise, it doesn't undermine the distributed TTCB operation.

Middleware

The middleware layers implement functionality at different levels of abstraction and make it accessible at the interfaces of several middleware modules. These interactions occur through the runtime environment via predefined APIs.

As mentioned earlier, a middleware layer can overcome the fault severity at lower layers and provide certain functions in a trustworthy way. A (distributed) transactional service, for example, trusts that a (distributed) atomic multicast component ensures typical properties (agreement and total order), regardless of the fact that the underlying environment can suffer malicious Byzantine attacks.

MAFTIA's intrusion-tolerance strategies

Given the variety of possible MAFTIA applications, different architectural strategies should be available to cope with different risk scenarios. MAFTIA offers several intrusion-tolerance strategies through a versatile combination of admissible failure assumptions. System designers can apply these strategies at several levels of abstraction in the architecture and, most important, in the implementation of the middleware and application services. An extended discussion appears elsewhere.[6]

Ultimately, MAFTIA supplies different solutions for different levels of threats and criticality (depending on the value of services or information), keeping the best possible performance-resilience trade-off. However, anything less than "arbitrary behavior" as an assumption raises eyebrows among many security and cryptography experts, so this statement deserves discussion.

A crucial aspect of any fault- or intrusion-tolerant architecture is the fault model on which the system architecture is conceived and component interactions are defined. Classically, making assumptions about hacker behavior isn't very sensible, which is why many system designers tend to assume any behavior is possible (asynchronous, arbitrary, or Byzantine).

However, such weak assumptions limit the system's power and performance—for example, could it still fulfill a service-level agreement (SLA), which is a contract a service provider makes with a client about quality of service (QoS)?

Architectural hybridization

Up until recently, increased system performance or QoS have meant less security. But MAFTIA has advanced the state of the art, demonstrating that it's possible to build applications that gather the best of both worlds: high resilience at the level of arbitrary failure systems and high performance at the level of controlled failure systems.

Through the innovative concept of *architectural hybridization*, the architecture simultaneously supports components with different kinds and severity of vulnerabilities, attacks, and intrusions.[7] For example, part of the system might be assumed to be subject to malicious attacks, whereas other parts are specifically designed in a different manner, to resist different sets of attacks. These hybrid failure assumptions are in fact enforced in specific parts of the architecture, by system component construction, and are thus substantiated. That is, in MAFTIA, trusting an architectural component doesn't mean making possibly naive assumptions about what a hacker can or can't do to that component. Instead, the component is specifically constructed to resist a well-defined set of attacks.

Wormholes model

Architecture isn't enough to solve the resilience problem, though. The correctness arguments of the algorithms and protocols rely on the *wormholes model*, a hybrid distributed-systems model that postulates the existence of enhanced parts (or wormholes) of a distributed system capable of providing stronger behavior than is assumed for the rest of the system. MAFTIA perfected this hybrid distributed-system model specifically for Byzantine faults.[7]

Protocol participants exchange messages in a world full of threats. If some of them are malicious and cheat, a wormhole can implement a degree of trust for low-level operations: as a local oracle whose information can be trusted, as a channel that participants can use to get in touch with each other securely (even for rare moments and for scarce bits of information), or as a processor that reliably executes a few specific functions or synchronization actions. Systems using strands of this model have received increasing amounts of attention lately.

A wormhole in the particular use of a trusted security component might look like a trusted computing base with a reference monitor. In fact, the concept is more general in two senses. First, trusted computing base's philosophy was based on system-level prevention of intrusions, whereas wormhole models have intrusion tolerance in mind: they would allow (and tolerate) intrusions even in the reference-monitor-protected part, significantly reducing the part of the system about which strong claims of tamper-proofness are made. Second, a wormhole can implement any semantics, including a reference monitor, but also simpler mechanisms, such as random number generators or key distribution, or innovative distributed agreement microprotocols.[8]

Real wormholes

The wormholes model can be realistically implemented via the notion of architectural hybridization.[7] MAFTIA implements each wormhole as a trusted–trustworthy component—a component that can be trusted because it's "better" by construction.

Figure 3 shows a snapshot of a real system that uses TTCB wormholes. The general, or payload, subsystems are the normal machines and networks depicted in dark shading in the figure. Each host contains the typical software layers such as the operating system, runtime environment, and middleware. Think of a small appliance board connected to the machine bus (these boards exist as COTS components), implementing a set of useful functions in a protected manner. This is the MAFTIA TTCB wormhole, depicted with light shading in Figure 3. This proof-of-concept prototype was built using simple prevention techniques. However, adequately designed MAFTIA wormholes can withstand extreme attack levels, even life-cycle attacks (insertion of malicious code during development), by resorting to recursive design: a wormhole can itself be a modular or distributed subsystem designed with intrusion-tolerance techniques (replication, diversity, obfuscation, rejuvenation), to substantiate trustworthiness claims as firmly as desired, such as eliminating single points of failure.

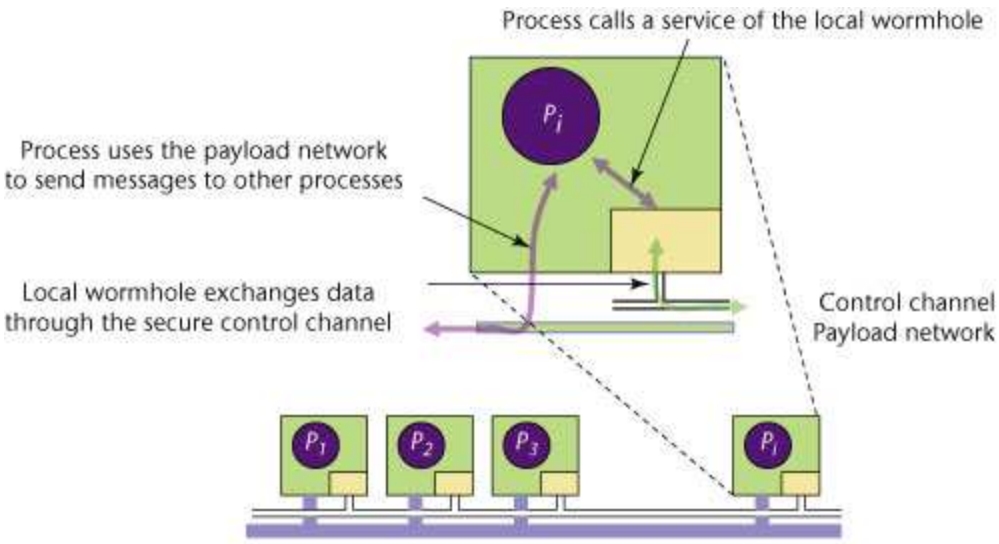

Figure 3. Architecture with a Trusted Timely Computing Base (TTCB) wormhole. The general systems are depicted in dark shading, and the wormhole is in light shading.

As we mentioned earlier, wormholes can be local or distributed. The TTCB is distributed through a control channel, which is a private network. As a practical example, consider a Web server farm in a data center that's tolerant to intrusions from the Internet: servers are connected through the payload Ethernet to the Internet, but the local wormhole boards are isolated from the directly attackable servers. The boards are interconnected through a secondary Ethernet that's completely isolated from the payload Ethernet or Internet—this is the private control channel. Even if the machine is corrupted, the hacker can't tamper with the local wormhole board or with the control channel.

Trusted–trustworthy components

MAFTIA assumes a fairly severe fault model, assuming that hosts and the communication environment are asynchronous and can all be intruded upon. However, hosts can have local trusted components implementing certain functions (such as random number generation, signature, and time) that can be invoked at certain steps of the MAFTIA software's operation and whose result can be trusted as always correct, regardless of intrusions in the rest of the system. The construction of the MAFTIA authorization service followed this local trusted components strategy, which is implemented around Java cards fitted in some hosts.[5]

The distributed trusted components strategy amplifies the scope of trust. As such, certain global actions can be trusted (such as global time and block agreement), despite generally malicious communication and host environments. MAFTIA implements this strategy through the TTCB, which is in effect a security kernel distributed across several hosts. Several of the MAFTIA middleware protocols follow this strategy, and in fact these protocols support the MAFTIA intrusion-tolerant transactional service.[6]

Arbitrary failure assumptions

The hybrid failure approach, however resilient, relies on trusted component assumptions, or trustworthiness. Several operations will have a value or criticality such that the risk of failure due to possible violation of these assumptions, however small, can't be incurred.

The only way to lower the risk even further is by resorting to arbitrary failure modes, in which nothing is assumed about the way components could fail. Consequently, this is another strategy pursued in MAFTIA—arbitrary-failure-resilient components—namely, communication protocols of the Byzantine class that don't make assumptions about the existence of trusted components. Some of the protocols the MAFTIA middleware uses to follow this strategy are of the probabilistic Byzantine class and offer several qualities of service (binary and multivalued Byzantine agreement and atomic broadcast). Some MAFTIA trusted-third-party services rely on them.[9]

MAFTIA middleware

Figure 4 shows the MAFTIA middleware's layers. The lowest layer is the *multipoint network* (MN), which is created on the physical infrastructure. Its main properties are the provision of multipoint addressing, basic secure channels, and

management communications, all of which hide the underlying network's specificities.

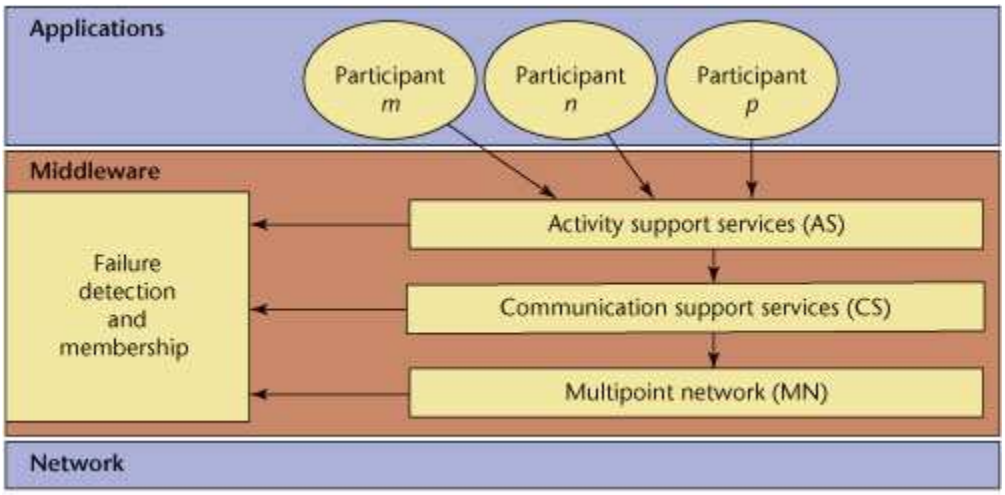

Figure 4. Detail of the MAFTIA middleware, showing the different architectural blocks.

The *communication support services* (CS) module implements basic cryptographic primitives, Byzantine agreement, group communication with several reliability and ordering guarantees, clock synchronization, and other core services. The CS module depends on the MN module to access the network. The *activity support services* (AS) module implements building blocks that assist participant activity, such as replication management, leader election, transactional management, authorization, key management, and so forth. It depends on the services the CS module provides.

The block to the left of the figure implements failure detection and membership management. Failure detection assesses remote hosts' connectivity and correctness and local processes' liveness. Membership management, which depends on failure information, helps create and modify group memberships (registered members) and the view (currently active, nonfailed, or trusted members). Both the AS and CS modules depend on this information.

As discussed earlier, an established way for achieving fault or intrusion tolerance is to distribute a service among a set of servers and then use replication algorithms for masking faulty servers. No single server has to be trusted completely, and the overall system derives its integrity from a majority of correct servers. Consequently, a very important part of the MAFTIA architecture is related to the algorithmic suites that implement communication and agreement among processes in different hosts.

Let's look more closely at the two main configurations of the MAFTIA middleware and algorithms.[6]

Byzantine agreement in an arbitrary world

In this configuration, the system model doesn't include timing assumptions and is characterized by a static set of servers with point-to-point communication and the use of modern threshold cryptography. There are no a priori trusted components, and trusted applications are implemented by deterministic state machines replicated on all the servers and initialized to the same state. An atomic broadcast protocol delivers client requests, imposing a total order on all of them and guaranteeing that the servers perform the same sequence of operations. The atomic broadcast is built from a randomized consensus protocol—that is, a protocol for Byzantine agreement.

Model. This asynchronous model is subject to Fischer, Lynch, and Paterson's[10] impossibility result of reaching consensus by deterministic protocols (FLP). Many developers of practical systems seem to have avoided this model in the past and built systems that are weaker than consensus and Byzantine agreement. However, randomization can solve Byzantine agreement in an expected constant number of rounds, as MAFTIA does. We use Byzantine agreement as a primitive for implementing atomic broadcast, which in turn guarantees a total ordering of all delivered messages.

Cryptography. To protect keys, we use threshold cryptography, an intrusion-tolerant form of secret sharing. Secret sharing lets a group of n parties share a secret such that t or fewer of them have no information about it, but $t + 1$ or more can uniquely reconstruct it. However, someone can't simply share a cryptosystem's secret key and reconstruct it to decrypt a

message because as soon as a single corrupted party knows the key, the cryptosystem becomes completely insecure and unusable.

In a *threshold public-key cryptosystem*, for example, each party holds a *key share* for decryption, which is done in a distributed way. Given a ciphertext resulting from encrypting a message, individual parties decrypting it output a decryption share. At least $t + 1$ valid decryption shares are required to recover the message. Another important cryptographic algorithm is the *threshold coin-tossing scheme*, which provides a source of unpredictable random bits that only a distributed protocol can query. It's the key to circumventing the FLP impossibility result, and the randomized Byzantine agreement protocol uses it in MAFTIA.

No timing assumptions. Working in a completely asynchronous model is attractive because the alternative of specifying timeout values actually constitutes a system's vulnerability. It's sometimes easier, for example, for a malicious attacker to simply block communication with a server than subvert it, but a system with timeouts would nevertheless classify the server as faulty.

This is how attackers can fool time- or timeout-based failure detectors into making an unlimited number of wrong failure suspicions about honest parties. The problem arises because a crucial assumption underlying the failure detector approach—namely, that the communication system is stable (failure detection is accurate) for a sufficiently long period to allow protocol termination—doesn't hold against a malicious adversary.

Secure asynchronous agreement and broadcast. Several protocols are used in this architecture configuration, such as reliable and consistent broadcast, atomic broadcast, and secure causal atomic broadcast. Detailed descriptions appear elsewhere.[9,11,12] These protocols work under the optimal assumption that fewer than one-third of the processes become faulty at any time. They're implemented in a modular and layered way.

Byzantine agreement requires all parties to agree on a binary value proposed by an honest party. Our randomized protocol checks if the proposal value is unanimous or else adopts a random value.[11] Multivalued Byzantine agreement is based on the previously described protocol and provides agreement on values from large domains.[12] A basic broadcast protocol in a distributed system with failures is reliable broadcast, which provides a way for a party to send a message to all other parties. It requires that all honest parties deliver the same set of messages and that this set includes all messages broadcast by such parties, but makes no assumptions if a message's sender is corrupted. An atomic broadcast guarantees a total order on messages such that honest parties deliver messages in the same order. Any implementation of atomic broadcast must implicitly reach agreement whether to deliver a message sent by a corrupted party, and, intuitively, this is where the Byzantine agreement module is needed: the parties proceed in global rounds and agree on a set of messages to deliver via multivalued agreement.[12] A secure causal atomic broadcast ensures a causal order among all broadcast messages. It's implemented by combining atomic broadcast with a robust threshold cryptosystem. Encryption ensures that messages remain secret up to the moment at which they're guaranteed to be delivered, preventing any violations of causal order by a corrupted party.

Reliable communication

Other MAFTIA middleware configurations follow a strategy based on distributed trusted components and architectural hybridization. In this configuration we implemented several distributed protocols, such as reliable multicast, atomic multicast, and consensus. These protocols' correctness arguments rely on the wormholes model.

Model. In this configuration, a group of processes executes a protocol, as Figure 3 suggests. Processes run outside the wormhole (in the dark part) and communicate by sending messages through the payload network. At certain points of their execution, however, they can request trusted services from the wormhole by calling its interface.

The global system assumptions in this configuration are weak: the system is assumed to be asynchronous, and processes and communication can suffer Byzantine faults. Consequently, this model is also bound to the FLP impossibility result[10] mentioned earlier. The wormholes model lets processes invoke functions that have enough power to circumvent the FLP impossibility, while still maintaining a generically weak and thus resilient model.[7,13] In this case, the TTCB and the control channel provide timely (synchronous) execution and communication among TTCB modules. In practical implementations, these synchrony guarantees can be ensured (despite the rest of the system being completely asynchronous) because the wormhole has complete control over its resources. Furthermore, it's assumed to fail only by crashing: it either provides its services as expected, or it simply stops running.

Example TTCB wormhole services. In MAFTIA, the wormholes metaphor is materialized by the TTCB, whose most important services are the local authentication service, which makes the necessary initializations and authenticates the local wormhole component before the process; the trusted time-stamping service returns timestamps with the current global time; and the trusted block agreement service applies an agreement function to a set of values and returns information about who proposed what.

Designing wormhole-aware protocols. In the project, we designed several protocols to form a coherent Byzantine-resilient communication suite, comprising reliable multicast,[14] atomic multicast, simple,[8] and vector consensus,[13] and state machine replication management.[15]

A correct use of the wormholes principle mandates that most of the protocol execution occurs in the payload subsystem. The wormhole services are only invoked when there is an obvious trade-off between what is obtained from the wormhole service and the complexity or cost of implementing it in the payload subsystem.

As a quick example, let's assume a reliable broadcast execution. The protocol starts with the sender multicasting a message through the payload channel; now the message's integrity and reception must be checked. In classical protocols, this entails some complexity or delay because the sender might be malicious or the network might be attacked or have omission failures. Wormholes can help here: the sender and all recipients send a hash of the message to the wormhole, which runs a simple agreement on the hashes in its protected environment, returning to everyone the sender's hash as a result. If all goes well, the protocol terminates in an extremely quick manner. In case of faults, additional information returned by the wormhole allows fast termination after a few additional interactions.

One achievement related to our model is that most of the protocols have lowered known bounds on the required total number n of processes to tolerate a given number of Byzantine faults f, from $n > 3f$ to $n > f + 1$ for reliable multicast14 and $n > 2f$ for state-machine replication.[15]

Another achievement concerns performance and complexity. Despite maintaining Byzantine resilience and working on essentially weak arbitrary and asynchronous settings, the protocols—thanks to wormholes—exhibit unusually high performance and low complexity when compared to alternative implementations.

You can find a detailed description of MAFTIA's history at http://istresults.cordis.lu/index.cfm/section/news/tpl/article/BrowsingType/Features/ID/69871.

Acknowledgments

MAFTIA was a project of the EU's IST (Information Society and Technology) program. The EC supported this work through project IST-1999-11583.

References

1. P. Veríssimo, N.F. Neves, and M. Correia, "Intrusion-Tolerant Architectures: Concepts and Design," *Architecting Dependable Systems*, *LNCS 2677*, R. Lemos, C. Gacek, and A. Romanovsky, eds., Springer-Verlag, 2003, pp. 3–36.
2. D. Powell and R.J. Stroud, eds., *Conceptual Model and Architecture of MAFTIA*, Project MAFTIA Deliverable D21, Jan. 2003. (pdf)
3. R. Stroud et al., "A Qualitative Analysis of the Intrusion-Tolerant Capabilities of the MAFTIA Architecture," *Proc. Int'l Conf. Dependable Systems and Networks* (DSN 04), 2004, pp. 453–461.
4. M. Correia, P. Verissimo, and N.F. Neves, "The Design of a COTS Real-Time Distributed Security Kernel," *Proc. 4th European Dependable Computing Conf.*, Springer Verlag, 2002, pp. 234–252.
5. Y. Deswarte et al., "An Intrusion-Tolerant Authorization Scheme for Internet Applications," *Proc. Int'l Conf. Dependable Systems and Networks*, IEEE CS Press, 2002, pp. C1.1–C1.6.
6. P. Veríssimo et al., "Intrusion-Tolerant Middleware: The MAFTIA Approach," tech. report DI/FCUL TR 04–14, Dept. of Informatics, Univ. of Lisbon, Nov. 2004.
7. P. Verissimo, "Uncertainty and Predictability: Can They Be Reconciled?" *Future Directions in Distributed Computing*, *LNCS 2584*, Springer-Verlag, 2003, pp. 108–113.
8. N.F. Neves, M. Correia, and P. Verissimo, "Solving Vector Consensus with a Wormhole," *IEEE Trans. Parallel and*

Distributed Systems, vol. 16, no. 12, 2005, pp. 1120–1131.

9. C. Cachin and J.A. Poritz, "Secure Intrusion-Tolerant Replication on the Internet," *Proc. Int'l Conf. Dependable Systems and Networks*, IEEE CS Press, 2002, pp. 167–176.
10. M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, 1985, pp. 374–382.
11. C. Cachin, K. Kursawe, and V. Shoup, "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography," *Proc. 19th ACM Symp. Principles of Distributed Computing*, ACM Press, 2000, pp. 123–132.
12. C. Cachin et al., "Secure and Efficient Asynchronous Broadcast Protocols," *Advances in Cryptology, LNCS 2139*, J. Kilian, ed., Springer-Verlag, 2001, pp. 524–541.
13. M. Correia et al., "Low Complexity Byzantine-Resilient Consensus," *Distributed Computing*, vol. 17, no. 3, 2005, pp. 237–249.
14. M. Correia et al., "Efficient Byzantine-Resilient Reliable Multicast on a Hybrid Failure Model," *Proc. 21st IEEE Symp. Reliable Distributed Systems*, IEEE CS Press, 2002, pp. 2–11.
15. M. Correia, N.F. Neves, and P. Verissimo, "How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems," *Proc. 23rd IEEE Symp. Reliable Distributed Systems*, IEEE CS Press, 2004, pp. 174–183.

Paulo E. Veríssimo is a professor at the University of Lisbon Faculty of Sciences and director of the LASIGE research laboratory. His research interests include architecture, middleware, and protocols for distributed, pervasive, and embedded systems, in the facets of real-time adaptability, security, and fault and intrusion tolerance (http://navigators.di.fc.ul.pt/).

Nuno F. Neves is an assistant professor at the University of Lisbon, Portugal. His research interests include security and fault tolerance in parallel and distributed systems. Neves has a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the IEEE.

Christian Cachin is a research staff member at the IBM Zurich Research Laboratory. His research interests include cryptographic protocols, security in distributed systems, and steganography. Cachin has a PhD in computer science from ETH Zürich. He is a member of the IACR, ACM, and IEEE.

Jonathan Poritz is a lecturer at Colorado State University. His research interests include mathematical cryptography,quantum computing, privacy technology, trusted computing, and computer visualization of geometry. Poritz has a PhD in mathematics from the University of Chicago. He is a member of the ACM and the American Mathematical Society.

David Powell is a "directeur de Recherche CNRS" at LAAS-CNRS. His research interests include distributed algorithms for software-implemented fault-tolerance, ad hoc networked systems, and critical autonomous robotic systems. Powell has a PhD in computer science from the Toulouse National Polytechnic Institute.

Yves Deswarte is a "directeur de recherche" at the CNRS Laboratory for Analysis and Architecture of Systems. His research interests include fault-tolerance, security, and privacy in distributed computing systems.

Robert Stroud is a reader in computing science at the University of Newcastle upon Tyne. His research interests include security and fault-tolerance. Stroud has a PhD in computing science from University of Newcastle upon Tyne.

Ian Welch is a lecturer at Victoria University, New Zealand. His research interests include secure auctions, intrusion detection, aspect-oriented development, and community computing. Welch has a PhD from the University of Newcastle upon Tyne.

Related work in intrusion tolerance

One of the first architectural attempts to build a secure and robust architecture was Delta-4, a system developed in a European project in the 1980s.[1] It provided distributed intrusion-tolerant services for data storage, authentication, and authorization.

More recently, several projects have also addressed this problem, providing secure middleware,[2] secure group or broadcast communication,[3] or secure authentication.[4] Oasis is a large US program on intrusion-tolerance research comparable to MAFTIA.[5]

References

1. D. Powell et al., "The Delta-4 Approach to Dependability in Open Distributed Computing Systems," *Proc. 18th IEEE Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, 1988, pp. 246–251.
2. M.K. Reiter, "Distributing Trust with the Rampart Toolkit," *Comm. ACM*, vol. 39, no. 4, 1996, pp. 71–74.
3. Y. Amir et al., "Secure Group Communication in Asynchronous Networks with Failures: Integration and Experiments," *Proc. 20th IEEE Int'l Conf. Distributed Computing Systems*, IEEE CS Press, 2000, pp. 330–343.
4. L. Zhou, F. Schneider, and R. van Renesse, "COCA: A Secure Distributed On-Line Certification Authority," *ACM Trans. Computer Systems*, vol. 20, no. 4, 2002, pp. 329–368.
5. J.H. Lala, ed., *Foundations of Intrusion Tolerant Systems*, IEEE CS Press, 2003.

Cite this article:
Paulo E. Veríssimo, Nuno F. Neves, Christian Cachin, Jonathan Poritz, David Powell and Yves Deswarte, Robert Stroud, and Ian Welch, "Intrusion-Tolerant Middleware: The Road to Automatic Security," *IEEE Security & Privacy*, vol. 4, no. 4, July/August 2006, pp. 54-62.