Integrating End-to-end Security and Fault Tolerance

Andrew C. Myers
Computer Science Department
Cornell University
andru@cs.cornell.edu

1 Trustworthiness by construction

A *trustworthy* system should tolerate both benign failures and malicious attacks, while providing confidentiality, integrity, and availability. That is, it should keep sensitive data confidential, resist data corruption, and remain available despite attacks and failures.

Many useful mechanisms have been developed for building trustworthy distributed systems: for example, encryption, digital signatures, replication, access controls, capabilities, and various distributed protocols. Having built a system using these mechanisms, how does the designer know that the desired trustworthiness goals have been achieved? There are several reasons why this is difficult. First, there are no established techniques for specifying all the aspects of trustworthiness or for showing that these specifications are satisfied. Second, distributed systems are complex and it is difficult to reason about what might happen in the presence of a malicious attacker or of failures. Finally, distributed systems in general must enforce the security of more than one participant, and these participants might not trust one another.

While current work on validating secure protocols continues to make progress, scaling this approach up to complete software implementations does not seem tractable because techniques based on exhaustive search do not scale. Further, analyses of communications protocols alone do not give assurance that the system as a whole is trustworthy.

A possible alternative is to build systems that are trustworthy *by construction*. The idea is that programs are written with explicit annotations specifying high-level requirements for confidentiality, integrity, and availability. Using these annotations, a compiler statically analyzes the program and, if necessary, transforms the program into a distributed system that performs the original computation in a trustworthy way. The program transformation automatically composes standard distributed system implementation techniques, driven by these explicit trustworthiness annotations.

Each principal explicitly expresses a certain degree of trust in the host machines that are used for computation and storage. Program trustworthiness policies (which apply to data and computation) are also *owned* by principals, so principals can disagree about how trustworthy the data, computation, and hosts in the system need to be. If components of the system fail—perhaps maliciously—the damage that can be caused is strictly bounded. A policy captured in a program annotation can be violated only if some host that is trusted by the owning principal to enforce that policy has failed.

This approach has been demonstrated for enforcing security properties such as confidentiality and integrity [ZZNM02, ZCMZ03], using underlying techniques that include partitioning, replication, encryption, capabilities, access control, and commitment. Because the compiler instantiates these mechanisms in response to explicit security policy annotations, the validation problem is lifted from the program to the compiler that transforms it. The same approach may be able to enforce certain kinds of availability as well, resulting in a single enforcement mechanism that gives assurance for all three major aspects of trustworthiness. This paper sketches some ideas in this direction.

2 Dependency and end-to-end policies

End-to-end trustworthiness properties constrain the end-to-end behavior of the system and therefore place stronger constraints on system behavior than local mechanisms such as access control. For example, access control can mediate access to a file, but doing so offers no guarantees about whether the information in the file is subsequently leaked. Access control is useful, but in large systems deciding how to set access control permissions becomes a difficult problem in itself (and one that end-to-end security analyses can help with).

In developing mechanisms such as access control, the security community has often ignored the end-to-end view of trustworthiness, whereas the fault tolerance community has typically had an end-to-end view. This difference in perspective seems to arise because security can often be phrased as a safety property [Sch01], whereas availability cannot. Thus, for many security properties, it is possible to halt the system when a security violation is about to occur. Clearly the strategy of halting the system is not helpful when an availability violation is about to occur, because this violates the very policy that is to be enforced!

However, there is also some commonality in these two notions of trustworthiness. To reason about whether a system is trustworthy in an end-to-end sense, it seems to be necessary to understand how data and computation depend on results computed earlier. The ultimate goal is to show that the observable output (and whether there is output) depends both on inputs and on the trustworthiness of host machines only in ways permitted by the security and availability specification.

3 Information flow

In the security domain, information flow analysis can be used to reason about end-to-end confidentiality and integrity. For example, to ensure that information remains confidential, it is necessary to prevent improper *information flows*. Information flow is simply a form of dependency. If a high-level input influences a low-level output, then information can be potentially learned from a change to that output.

Information flow control has been enforced for both confidentiality and integrity properties with increased precision using a variety of type systems and static dependency analyses (e.g., [DD77, VSI96, ML97, ZZNM02, PS02, BN02]). An example of this approach is the Jif language [Mye99, MZZ+03], which extends Java with security annotations. In Jif, every value has a *security type* with two parts: an ordinary type such as `int`, and a sensitivity label that describes how the value may be used. Any type expression *t* may be decorated with any label expression {*l*}; the resulting type expression is written as *t*{*l*}. A label defines an information flow policy on the use of the labeled data. Security types make security policies explicit in the system design, making these decisions auditable and verifiable.

Labels in Jif are defined by the *decentralized label model* (DLM) [ML98, ML00]. For example, the label {p: q} describes information that principal p owns and principal q can read. One label L_1 may be at least as restrictive as another label L_2, if the restrictions on the use of data labeled by L_1 are at least as strong as those on data labeled by L_2. This relationship is denoted formally as $L_2 \sqsubseteq L_1$; it indicates the direction in which information may securely flow. For example, we have {p: q,r} $\sqsubseteq$ {p: q} because the label {p: q} allows fewer readers.

4 Availability policies

Dependency analyses can identify potential availability vulnerabilities too. If computation A depends on results from computation B, but those results are unavailable, then A must also be unavailable. This is the same kind of analysis that is needed to analyze whether confidential information flows from B to A, or whether low-integrity information does.

For example, an availability policy A_1 on data might demand that the data be available as long as the hardware hasn't suffered a random failure and the site password file has not been compromised. Because it allows more failures, this is a weaker availability policy than a policy A_2 that guarantees the data available

only as long as the hardware hasn't failed. Because the implication $A_2 \Rightarrow A_1$ holds, it is acceptable for a program to assign more-available data labeled A_2 into a less-available location labeled A_1. Thus, the statement $A_2 \sqsubseteq A_1$ holds when the policies are interpreted as information flow labels. (The syntactic form of policies A_1 and A_2 is deliberately left abstract.)

Integrity and availability are superficially similar, but they promise different things. If data has integrity, then that data will be correct—if it is provided. If it is available, then it will be provided—but it might not be correct. Integrity and availability also behave differently in the presence of replication. Suppose there are two sources 1 and 2 of the same data, with integrities I_1 and I_2 and availabilities A_1 and A_2. Then reading both data values and proceeding only if they agree increases integrity (to $I_1 \sqcap I_2$) but *decreases* availability (to $A_1 \sqcup A_2$), because both replicas must be available to do the check. (Here $\sqcap$ and $\sqcup$ are respectively the greater lower bound and least upper bound for the order $\sqsubseteq$.) Thus, a more accurate analysis of the trustworthiness of replicated computation is obtained by treating integrity and availability separately. More complex protocols can also be analyzed similarly, though it is not yet clear how far this approach can be pushed.

The decentralized label model can be extended so principals can express availability policies. We can then ask whether a system satisfies the availability policies of all principals with a stake in the security of the computation. The insight is that principals on the right-hand side of a DLM policy can be interpreted as modes of failure (where a failure may result from a successful attack). A confidentiality policy $\{p\!:\!q\}$ is an assertion that p believes the data will remain confidential as long as q does not fail to keep it confidential. For integrity, p believes the data will have integrity unless q fails to provide correct data. As an availability label, it says that p believes that the data is available as long as failure q does not occur. In each case the policy component q can be interpreted as a failure that may compromise some aspect of security.

It is also possible to capture the independence or dependence of different possible failure modes through a conjunction operator ($\wedge$) that operates on failures. The failure $q_1 \wedge q_2$ represents the simultaneous failure of q_1 and q_2. If $q_1 \wedge q_2 = q_1$, then the failure of q_1 is always accompanied by the failure of q_2. By defining a suitable algebra over failures, many different failure models can be represented in this label system. For example, independence of failures is captured by a conjunction operator that simply multiplies failure probabilities when appropriate.

5 Conclusions

The extended label model sketched above provides a basis for specifying all three major aspects of trustworthiness: confidentiality, integrity, and availability. Program dependency analysis can provide an end-to-end enforcement mechanism for all three aspects. And there is an appealing way to satisfy security policies by rewriting programs onto a distributed system. Integration of these two threads is clearly the next step to take: a compiler back end that can rewrite programs to enforce availability, using not simple replication but more complex distributed protocols that can improve both integrity and availability in the presence of Byzantine failure. If this approach works, the result will be a unified way to build distributed systems that are trustworthy by construction.

Acknowledgments

I would like to thank Lantian Zheng and Lorenzo Alvisi for many interesting discussions on availability policies and on enforcing availability, and Fred Schneider for many suggestions on an earlier version of this text.

References

[BN02] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2002.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[ML98] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, CA, USA, May 1998.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[MZZ+03] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2003.

[PS02] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[Sch01] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[ZCMZ03] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 236–250, Oakland, California, May 2003.

[ZZNM02] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.