Trust Distribution Diagrams: Theory and Applications

Michael E. Locasto *locasto@ucalgary.ca*

Steven J. Greenwald *sjg6@gate.net*

Sergey Bratus *sergey@cs.dartmouth.edu*

Abstract

Software systems have rapidly increased in complexity, making it difficult to argue about their trustworthiness in prose. It also remains difficult to construct and maintain a formal proof of security for complex, evolving systems. For example, given the design of two software systems, we currently lack a principled method for discerning which system we should regard as the more *trustworthy* (e.g., containing a simpler or more cohesive set of trust relationships). *Current metrics (e.g., lines of code) have little relationship with the actual strength of the code or design.*

This paper introduces the notion of *diagramming* trust relationships in a system using a novel construct we call Trust Distribution Diagrams (TDDs). TDDs define a graphical language for expressing the distribution, amount, and migration of trust in design–level components. We suggest that TDDs can help those engaged in the craft of security architecture formulate understandable layered assurance arguments.

1 Introduction

Assurance arguments, particularly for the composition of commodity software systems, remain difficult to achieve. Crafting an assurance argument requires more than just prose; at the same time, overly complicated formalisms can prove difficult to maintain. Furthermore, abstract models of system design and behavior targeted for formal analysis typically find themselves divorced from the assumptions, limitations, and properties of the running code. Thus, these models sometimes have little to say about the actual level of trustworthiness of a running piece of code in a specific computing environment. As a result of these difficulties, software developers for commodity software applications (e.g., popular Web browsers, email clients, document readers, multimedia viewers, chat programs) typically eschew the creation of comprehensive assurance arguments as either not worthwhile (because they would not match a very complicated reality) or not worth the effort (because the time involved in creating a logical, consistent, and rigorous line of reasoning that illustrates the security properties of their system would far outstrip the time spent on writing code or maintaining the system).

1.1 Contribution

We personally value assurance arguments, but we also acknowledge that such arguments can prove difficult to formulate for a variety of reasons (lack of training or practice included). We instead suggest that developers can construct a qualitative argument about software trustworthiness based on precise, accurate diagrams that capture and convey the amount and distribution of trust among a subset of system components, including trust relationships and dependencies, along with the relative risk inherent in these relationships. We note that the relationship of trusted components, their communication paths, and their data dependencies moderates *design* risk. Although errors will always exist in system implementations, it seems feasible to rigorously describe the distribution and amount of trust in design-level components in a manner both visual and amenable to a structured (but not "formal") treatment.

The main contribution of this paper stems from *identifying a need*; we freely admit that we have yet to arrive at an exact formulation and design of this construct, but we see this workshop as a way to gather input from the community of practitioners and researchers engaged in constructing layered assurance arguments.

2 Motivation: Rapid Growth in Size and Complexity

"There are two ways to design a system. One is to make it so simple there are obviously no deficiencies. The other is to make it so complex there are no obvious deficiencies." — C. A. R. Hoare

Although in the early stages of computing, systems might have remained simple enough to formally verify or manually certify, most systems we currently rely on contain too much complexity (and change too rapidly) for such a process to have much practical success. Computer and information systems (including both security systems as well as application software we attempt to defend) have rapidly increased in complexity. At least one common example involves the growth in size of the Linux kernel; early releases of Linux boasted "only" 200,000 lines of code. Linux now approaches 10 million lines of code,[1] and some web browsers easily contain double or triple this number. Ironically, researchers have resurrected the same argument (size of an OS kernel relative to a "hypervisor") to support the trustworthiness of hypervisor-based application sandboxing.[2]

For most non-trivial software systems (e.g., web servers, browsers, media players, libraries, databases), analyzing the trustworthiness of this code or providing a correct proof of security can present a daunting (if not impossible) task. Arguments of trustworthiness based on small code size (as often done with hypervisors) ring no more true than for similar arguments about OS kernels or the kinds of software mentioned above. We simply lack the tools to do this — and the counter-example of the recent seL4 microkernel[3] verification effort [10, 11, 21] provides the exception that proves the rule: however noteworthy the results or impressive the effort might seem to us, a team of experts took months to construct the proof infrastructure for one specialized piece of software. In essence, systems have grown beyond our current capability to argue meaningfully about their reliability in English prose.

We commonly witness justifications that generally follow a line of reasoning similar to this claim (abstracted to protect the guilty): *"Our hypervisor is only 394 lines of C code and 22 lines of inline assembly. Because this TCB is very small, we have confidence that it is correct and bug free. The rest of the system may contain bugs, but all the security and assurance lie in this small TCB, so the entire system is secure."* This issue particularly relates to research efforts that make arguments about the trustworthiness of various security systems based on virtual machine and hypervisor architectures. The systems security community has borrowed a slew of VMMs, hypervisors, virtual machines, para-virtualized execution, and virtual servers, each promising to enforce isolation and increase the security of code running within them (claims subject to a healthy dose of skepticism [3] in certain scenarios).

Although few security practitioners would normally question the notion that a small TCB provides a "good" TCB, **we suggest that the relationship between size and trustworthiness remains *extremely* ill-defined**. We claim that metrics like lines of code (LOC) say little about the trustworthiness of a system, except in the sense that "ridiculously large" usually equals "definitely untrustworthy." All things equal, smaller might seem better, but on the other hand, we could trivially construct a very secure chunk of code with lots of LOCs — not to mention insecure code with few LOCs. The argument used to support the notion of "smaller as better" really boils down to "more tractable" (whether for human or automated analysis). And yet, the relationship between tractability and assurance depends mainly on the nature and structure of the codebase and the *type of analysis* done, not the raw size of the codebase considered in LOCs. In other words, the nature and structure of the codebase will determine the complexity (i.e., cost) of the analysis when applied to the code. For example, an otherwise large but straightline code with few decision structures and little or no looping may prove easy to validate automatically even though there are hundreds of thousands of lines of code. On the other hand, nested looping with multiple branches of decision control or significant modifications of data may quickly explode the state space for automated analysis even if the code base contains only a few hundred lines of code.

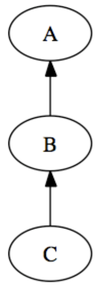
$$\begin{tikzcd} A \\ B \arrow[u] \\ C \arrow[u] \end{tikzcd}$$

Figure 1: *A Simple Hierarchy With 3 Trust Levels*. Here, C trusts B and B trusts A.

[1] http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php

[2] "The numbers: Xen hypervisor has around 200k LOC, while Linux/Windows/Mac kernels have tens of millions. Get it?" –http://theinvisiblethings.blogspot.com/2010/08/skeletons-hidden-in-linux-closet.html

[3] http://l4hq.org/

Although it might seem that researchers have a better chance at formally verifying small pieces of C code, a large gap remains between the mere existence of this possibility and strong, justifiable assertions about trust in an overall system design. Furthermore, the units of "size" remain unclear: lines of code? Number and *type* of components (i.e., a TPM plus N lines of kernel code)? Number of components that the TCB communicates with? Number of functions in an API or interface? Something else? In addition, the relative size of two TCBs for different tasks (or for the same task *across platforms*) has no meaning.[4] Here again we see that size fails to provide any meaningful indication of trustworthiness.

Yet, the community still subscribes to the sage advice that a small TCB supplies the principled cure for realizing a real-world reference monitor. Few qualifiers on "small" accompany this advice. We suggest an alternative: perhaps a better measure of assurance should rely on the **complexity of the trust relationships between system components** rather than the size of the codebase.

3 Trust Distribution Diagrams: Principles and Structure

We selected the name "Trust Distribution Diagrams" to evoke three concepts: (1) **trust**, (2) **change and loose binding** of trust in and between components, and (3) some **structured means of representing these concepts** such that both scientists and developers can make rigorous, objective comparisons between two or more designs.

We assert that "trust" and "trustworthiness" have distinct meanings and represent different concepts. Postulating a precise definition for both terms can present difficulty; in this context, we equate "trustworthy" with the notion that software "follows expected behavior" according to some security policy (where "behavior" consists of sequences of events that read or modify specific data structures).[5] In contrast, "trust" represents the concept of human judgement (warranted or not) of the potential reliability (or level of risk) in something. We contend that we can extend this notion of trust to express a relationship between non-human system components. Humans can trust something that may not exhibit trustworthiness, and they may distrust something that does in fact provide trustworthy behavior. Finally, humans may trust something when it displays a certain threshold level or history of trustworthiness. Different trust models can exist, such as *implicit trust*, *directed trust* (i.e., mandated trust regardless of trustworthiness), and *verified trust* (i.e., some certification or verification procedure has assessed the trust relationship). TDD semantics should capture these different trust model types.

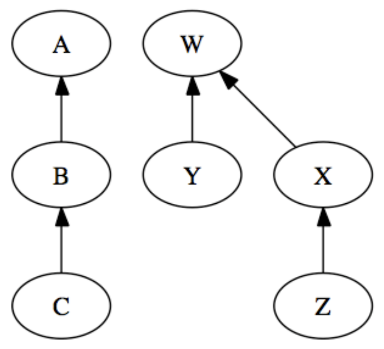

Figure 2: *Complex Hierarchy With 7 Trust Levels*. Note incommensurate trust of some levels (e.g., (B,X), (A,Z)).

3.1 Trust Policy = Trust Statements + Consequences

Intuitively, TDDs should encode statements about trust relationships. We believe that such trust statements should capture: (1) the subjects and objects involved, (2) the predicates they check on pre- and post-conditions, and (3) the temporal expectations or assumptions. For precision, it may help to formulate such statements (the precursors to illustrating or describing trust relationships) in E-Prime [5].[6] Let us start with a few simple examples:

1. *System A trusts System B to check property P at time T.*

[4]While some readers may find this point obvious, the notion of a TCB often gets casually discussed as if some universal TCB existed involving a specific version of the IA-32 microarchitecture, a specific version of the `x86` assembly language produced by a specific version of the Microsoft Visual C++ compiler running on a standard WindowsXP OS image.

[5]To tie this definition to "security": attacks attempt to violate these expected behaviors and can lead to security breaches (e.g., data exfiltration, malcode execution).

[6]The E-prime language removes all forms of the verb "to be" from standard English. Side effects of this change can include the specification of an observer and the potential for more precise statements. E-prime does require developing a certain mindset, but does not present an insurmountable challenge to communicating; we wrote this paper largely in E-prime.

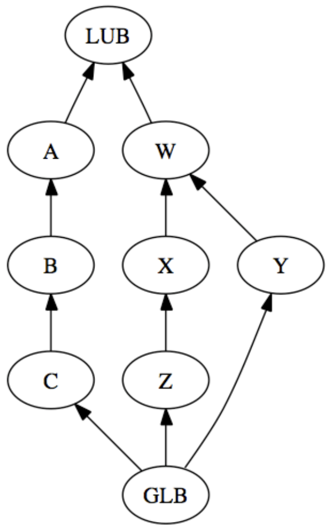

Figure 3: *TDD Lattice.* Least Upper Bound (LUB) corresponds to "totally trusted" and Greatest Lower Bound (GLB) corresponds to "totally untrusted." Note incommensurate levels (e.g., (A,Z), (A,Y), (A,W)).

2. *When Component B consumes input X, it requires that Component C not modify X after sequence S occurs.*
3. *The operating system trusts Component W at level I_i to raise identity to level I_{i+1}, acquire data structure* `port` *with a value of "80" and then resume identity level I_i.*

We can create *trust policy* by combining a trust statement with a consequence (e.g., "System A trusts System B to check property P at time T. If B fails to do so, A will henceforth check property P itself."). One important benefit of using TDDs comes from stating the *consequences* for violating expected behavior (note that we do not therefore need to predict *unexpected* behavior or imagine all the ways that the system might fail) and this provides flexibility. For example, in the last trust statement, if the OS does not see the event $\{$set port=80, resume $I_i\}$ (the expected behavior), then the OS has a range of options. It can chose one or a combination of terminating W, forcing the resumption of identity level I_i, denying access to 80, or reporting the violation of the policy.

3.2 Finding the Right Structure for Diagramming Relationships

Formulating the appropriate *structure* for TDDs as a standardized language for trust descriptions presents an important challenge. The underlying structure must convey the semantics (arrangement of concepts) regardless of whatever syntax (TDD symbols) we might eventually choose. **The essential feature of TDD structure comes from its ability to depict the "complexity" of the relationships it contains.** We need something both simple and expressive. A translation or transformation of Harel's "statecharts" [7] might provide a basis for representing TDD structure; Jackson Diagrams [8] provide another. Regardless, TDD structure should convey three key properties:

1. direction of trust relationships (*map* of trust relationships)
2. location of trust regardless of (current) level of trust (*orthagonality* of location and level)
3. how direction, location, and level change over time or in response to input or other events (*duration and migration* of trust)

Hence, we do not define "trust" as a static number, but rather as an evolving graph structure. Trust may move through subsets of system components (trust migration). It can increase and decrease under use and load conditions of those components, and it may change based on runtime modifications to the system. Transitive trust may exist (for now, we make no value judgement here).

The heart of a TDD centers on a graph of relationships between components of a system. Examples of components include humans, whole software systems, subsystems, components, input and output mechanisms, and data. Relationships include actions and predicates between components that express the nature of some form of composition; examples include: "trusts", "expects", "checks", "provides", "modifies" and "requires." TDDs will possess a unique feature that differs from most previous graph-based design frameworks (e.g., entity-relationship diagrams): they explicitely include temporal movement in placement and amount of trust.

A lattice may provide a natural way to express a trust relationship graph, particularly how trust flows from higher levels of trust to lower, essentially laying out a landscape of the system's components in terms of their trust levels (see Figure 3).

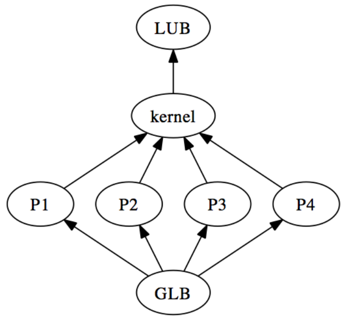

Figure 4: *Example lattice for a monolithic kernel.* A simple illustration of how processes trust the kernel.

4 Examples for Diagramming Trust

This section offers examples of different challenges in a variety of domains where system designers or evaluators might apply the concept of diagramming trust relationships. The theme underlying each of these examples stems from the fact that software developers have no way of expressing (and thus communicating to other developers, evaluators, or users) their trust assumptions about a piece of code. Here, we offer example TDDs for these scenarios. Since TDDs remain a developing concept the reader should treat these examples as a basis for discussion rather than definitive.

Example 2: AV cooperation/negotiation Anti-virus systems regularly interfere with each other [12, 19, 20] because they attempt to indiscrimminantly modify system properties and OS kernel data structures without coordinating. Note how the composition of integrity checkers can cause a degradation of availability. TDDs might prove useful here in defining a normalized order of approved modifications (independent of the problem of composing the policy needs or wants — often driven by market share considerations — of these two different pieces of software: a hard problem that this paper does not deal with). One interesting application of TDDs to the problem of providing some sort of negotiation framework (beyond trust management [2]) might actually offer an interesting way to put a number on the amount of additional "security" that an extra AV provides in terms of *coverage* of system properties and data paths.

Example 3: Variable Checking In a dynamically composed software system, where a chain of plug-ins checks or filters input, each individual plug-in may defensively check its arguments for sanity. Such "mutual distrust" represents a safe default, but it also entails potentially wasted work and a measurable impact on performance. Enabling a designer to express trust relationships between plug-ins may allow the designer or a runtime system to dynamically compose chains of components that trust the input or output of other components in the chain — and thereby reduce the "tax" that security measures so often impose.

Example 4: "GUI Requires User" Software applications often delegate security-sensitive decisions to end users (e.g., trusting an unrecognized or unsigned certificate, allowing a particular socket connection). This kind of delegation serves as the source of many instances of "human in the loop" vulnerabilities that can, for example, exploit the "trusted path" gap [22]. The human has little evidence with which to make a trust decision (see "Why Do Street-Smart People Do Stupid Things Online?" [4]). Often the application presents only a small GUI popup asking that some arcane technical action (e.g., "The application Google Maps is requesting a socket connection to IP 64.xxx.xxx.xxx") proceed. The GUI usually fails to present a binding between the requested action and the subsequent actions of the machine. Furthermore, the GUI also fails to indicate what *consequences* the user's refusal might result in. For example, it may happen that the user might obtain what they want with only a minor loss of information or functionality, but the GUI does not make this outcome known — the user effectively **must** acquiesce to the request for fear of frustrating their main goal. In addition, the GUI does not make clear what elevated privileges the GUI obtains on an affirmative response or under what conditions the GUI will relinquish those privileges. Greenwald's paper on E-Prime for security [5] contains a similar example involving ZoneAlarm phrasing of user notifications.

While we do *not* argue that TDDs provide a user-consumable way of solving the trusted path problem, we do claim that GUI designers can more easily understand the claims and trust decisions they require a user to make relative to a certain design approach. Too often software developers think of features first and security consequences last (if at all). Asking them to diagram the relationship between the system and the human user in terms of what the system expects of the user's trust may help them redesign the application to eliminate such easily exploited trust decisions or to clarify the trust conditions to the user.

5 Discussion: Using TDDs

Let us assume for a moment that TDDs exist and have a well–specified syntax and semantics. How might we use them as a security and assurance measurement tool?

5.1 Using TDD Complexity as an Evaluation Tool

Since the most obvious formulation of TDDs involves some type of graph, one might feel the temptation to use complexity measures rooted in the number of vertices and edges. We have a slightly different take: **TDDs are *patterns of trust* that may repeat in different contexts within the system, so understanding the entropy of the system in terms of these patterns gives an indication of the density or complexity of the TDD**. In other words, the similarity or "compressiblity" of the the set of trust relationships supplies a notion of diagram complexity (there remains the issue of how to classify the resulting score, please see below). For example, the graph in Figure 5 contains two distinct tokens even though it contains three connected components. The paths (and connected components) ABC and XYZ essentially have the same "shape" whereas the subgraph containing vertices $\{$D, E, F$\}$ has a different shape. Each graph shape can represent a "token" in terms of calculating entropy (to ease exposition, we make a simplifying assumption here that we can ignore the vertex labels for the purposes of identifying a trust pattern).

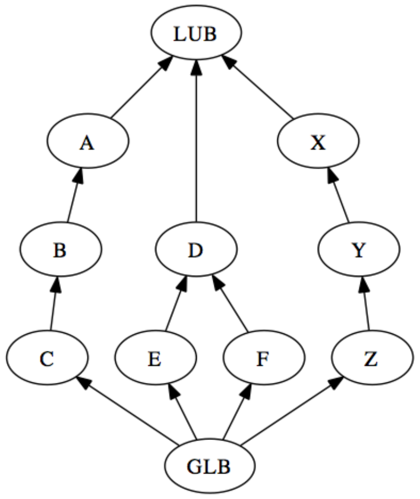

Figure 5: *Example Trust Graph*. This simple, static trust graph has two distinct patterns or tokens; its shape compression factor provides a possible method of ascertaining its relative trustworthiness to a similar design. This graph displays the *map* property, but it still lacks notation for both *orthagonality* and *duration/migration* (predicates controlling the *map*).

Recall that in the abstract we gave an example of the concept of "more trustworthy" as a system containing "a simpler or more cohesive set of trust relationships." The complexity of a system in terms of a large number of brittle trust dependencies (reflected in a correspondingly more complex TDD) might typically work to lower the overall level of trustworthiness of the design. Some problems, however, may demand a relatively complex TDD, given that trust might have to rest in different subsets of components at different points in time. Again, the intent behind using a TDD seeks to reveal trust complexity as a first step to reducing it. Claiming that one TDD "is better" than another depends on the definition of better:

1. a "less complex" TDD (perhaps given by the entropy of subgraph patterns)
2. a "more robust" TDD in that it contains redundancy and no single point of failure

3. a “checkable” TDD in that model checking can easily assess the structure of relationships with respect to a formal model
4. a “survivable” TDD that contains layers/rings of trust that it can jettison to save the core computation (this requires a definition of “core”)
5. a “nimble” TDD where trust migrates between components so that an attacker can never attack the whole set

In essence, an objective way of measuring TDDs for different systems uses a comparison of multiple timelines of how a particular set of trust relationships evolve. Note that the simple entropy measure suggested above does not take the “duration and migration” dimension of a TDD into account, but we can account for this by giving a vector of entropies, one for each system state.

5.2 Other Applications of TDDs

TDDs help evaluate system reliability in adversarial environments based on their structure, not the number of vulnerabilities removed or exploits frustrated.

For runtime systems capable of interpreting them, TDDs may offer a way to autonomically tolerate failure by gracefully reducing services or features as trusted components fail or become corrupted. In some sense, TDDs offer the notion of a “Trust TTL.” Such an application may require something akin to a tagged architecture, or a very well-labled (and mediated on a fine-grained scale) data and memory space. TDDs also might see use as a visualization technique that can show post-mortem how trust dependencies failed in response to an attacker subverting some set of components.

We intend that TDDs see use in the initial design of new, self-contained systems. Composing two legacy systems with separate TDDs might require a clean-slate TDD, although there may exist some clever composition tricks and properties. While we do not consider the secure composition problem (which tends to work as a totally binary relation), an interesting path of research might consider how to recast that problem in terms of varying levels of trust.

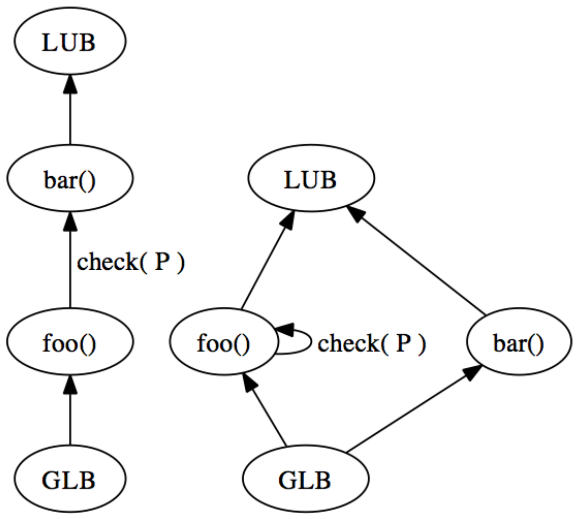

Figure 6: *Trust Evolution Over Time*. Here, the TDD expressed as a lattice, shows a change in the trust relationship between foo and bar.

5.3 Cavaets

We stress that we do not intend TDDs to support assertions like “System A is more secure than System B,” nor do we think that TDDs can prevent all errors in system implementation (unfaithful translation by either the human or compiler from the design to the artifact may result in unanticipated security flaws). Instead, we mainly see TDDs as providing a coherent basis for qualitative arguments about design–level trustworthiness; ultimately, the developer or designer may have misplaced this trust (but the design artifact visibly displays this placement rather than buries it in lines of code).

An interesting challenge exists in deciding how much to separate trust design from functional design. Two or more systems for accomplishing the same task may take radically different functional design paths. Since the unique features of their individual architectures may have an impact on trustworthiness, how can we create TDDs resiliant to such differences? In other words, how do we design TDDs so that we can still use them for trust comparisons even when the systems they describe have radically different designs?

6 Related Work

Assessing the trustworthiness of real-world systems designs presents a challenging task, particularly in terms of compliance checking [9, 1]. TDDs may provide a useful tool for such compliance tasks.

One of the closest concepts to TDDs is Harel's work on *state charts*; we intend to explore their application to this problem. State charts [7] have a number of attractive features that correspond to the needs we perceive for creating the underlying TDD semantics. State charts provide a method for expressing complex relationships in software systems in a compact fashion. They supply a visual discipline for understanding system complexity via *encapsulation* and predicated labeling of state transitions, and they also include a notion of history contained in each state.

The diagrams at the heart of Jackson Structured Programming [8] may serve (with appropriate modifcations) as another basis for TDD semantics related to describing trust relationships about data properties. Jackson Diagrams illustrate how a developer can design a system in terms of its data structures and its input and output formats. In this discipline, system designers graphically describe data formats in terms of sequences, iterations, and selections; the combination of input and output formats described in this way suggest the structure of a program for performing this mapping. For our purposes, the most important feature of Jackson Diagrams stems from their focus on data structure. While trust relationships may exist between processes in a system, the subject of this relationship often concerns an action on some data (e.g., component A expects component B to perform a predicate that tests a property of data D).

The transitive trust of the Unix rtools (rsh, rlogin, rcp) provides an early example of trust relationships between systems; in the network environment, understanding such relationships has become a standard tool in network design and analysis, particularly in terms of security. The body of work on attack graphs [18, 16, 13, 14, 15] supplies a method of codifying trust relationships in a structured graph representation.

Probst and Hansen [17] suggest the concept of *secure de-composition* (our term for their "fluid information systems" concept) whereby systems are designed to move data away from threatend environments. This kind of system highlights the principle of *trust distribution*: their system physically moves sensitive data to a more trustworthy set of components. They discuss a method of using static analysis and program partitioning to migrate applications and data during runtime.

Although similar to the concept of ER diagrams in software engineering, TDDs can be seen as a domain-specific language for specifying security-relevant expected behaviors in software relationships as they *evolve over time or in response to input conditions* (temporal relationships can be difficult to capture in traditional ER models [6]).

7 Conclusion

The information security community finds itself at a crossroads; despite intense efforts aimed at increasing the trustworthiness of software systems, errors, faults, and vulnerabilities still exist in deployed code. Few ways exist to quantitatively measure the relative amounts of security or trustworthiness in a system design. Almost all arguments about trustworthiness proceed qualitatively, under the guise of "argument from authorithy" (i.e., the personal opinion of an expert) or simple metrics (e.g., lines of code) that have little immediate relationship to the actual strength of the code or the design.

This paper suggests a method for giving structure to the *assumptions of a software architect, developer, or system designer about the expected behavior of a set of components*. The use of TDDs helps such expectations form the basis of a measure of relative trustworthiness in the relationships between data, components, and processes in the system. We conceive of TDDs as including techniques for representing the amounts and distribution of trust in system components and how these properties evolve over time or in response to input. Capturing these patterns and relationships in a concrete format seems like it can help make the trustworthiness properties of the system more explicit (and thus more amenable to structured, objective analysis).

References

[1] Adam Beautement, M. Angela Sasse, and Mike Wonham. The Compliance Budget: Managing Security Behaviour in Organisations. In *Proceedings of the New Security Paradigm Workshop*, 2008.

[2] Matt Blaze, Joan FeigenBaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the IEEE Conference on Security and Privacy*, May 1996.

[3] Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and Sean W. Smith. VM-based Security Overkill: A Lament for Applied Systems Security Research. In *Proceedings of the New Security Paradigms Workshop*, September 2010.

[4] Sergey Bratus, Chris Masone, and Sean W. Smith. Why Do Street-Smart People Do Stupid Things Online? *Security Privacy, IEEE*, 6(3):71–74, May 2008.

[5] Steven J. Greenwald. E-Prime for Security: a New Security Paradigm. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95. ACM, 2007.

[6] Heidi Gregersen and Christian S. Jensen. Temporal entity-relationship models-a survey. *IEEE Trans. on Knowl. and Data Eng.*, 11(3):464–497, 1999.

[7] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[8] Michael A. Jackson. *Principles of Program Design*. Academic Press, first edition, 1975.

[9] Klaus Julisch. Security Compliance: the Next Frontier in Security Research. In *Proceedings of the New Security Paradigm Workshop*, 2008.

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the* 22^{nd} *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[11] Rafal Kolanski and Gerwin Klein. Formalising the L4 microkernel API. In *CATS '06: Proceedings of the 12th Computing: The Australasian Theroy Symposium*, pages 53–68, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[12] Michael E. Locasto, Sergey Bratus, and Brian Schulte. Bickering In-Depth: Rethinking the Composition of Competing Security Systems. *IEEE Security and Privacy*, 7(6):77–81, 2009.

[13] Peng Ning, Yun Cui, Douglas Reeves, and Dingbang Xu. Tools and Techniques for Analyzing Intrusion Alerts. *ACM Transactions on Information and System Security*, 7(2):273–318, May 2004.

[14] Peng Ning and Dingbang Xu. Learning Attack Strategies from Intrusion Alerts. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security*, pages 200–209, October 2003.

[15] Peng Ning, Dingbang Xu, Christopher G. Healey, and Robert A. St. Amant. Building Attack Scenarios through Integration of Complementary Alert Correlation Methods. In *Proceedings of the* 11^{th} *Annual Network and Distributed System Security Symposium*, pages 97–111, February 2004.

[16] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. A Scalable Approach to Attack Graph Generation. In *Proceedings of the* 13^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2006.

[17] Christian W. Probst and Rene Rydhof Hansen. Fluid Information Systems. In *Proceedings of the New Security Paradigm Workshop*, 2009.

[18] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated Generation and Analysis of Attack Graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002.

[19] skape. What Were They Thinking: Annoyances Caused by Unsafe Assumptions. *Uninformed*, 1, April 2005.

[20] Skywing. What Were They Thinking: Anti-virus Software Gone Wrong. *Uninformed*, 4, June 2006.

[21] Harvey Tuch. Formal Verification of C Systems Code. *Journal of Autom. Reasoning*, 42(2-4):125–187, 2009.

[22] Zishuang (Eileen) Ye, Sean W. Smith, and Denise Anthony. Trusted Paths for Browsers. *ACM Transactions on Information and System Security*, 8(2):153–186, 2005.