Towards Usable Generation and Enforcement of Trust Evidence from Programmers' Intent

Michael Huth[1], Jim Huan-Pu Kuo[1], Angela Sasse[2], and Iacovos Kirlappos[2]

[1] Department of Computing, Imperial College London
London, SW7 2AZ, United Kingdom
{m.huth, jimhkuo}@imperial.ac.uk
[2] Department of Computer Science, University College London
London, WC1E 6BT, United Kingdom
a.sasse@cs.ucl.ac.uk

Abstract. Programmers develop code with a sense of purpose and with expectations on how units of code should interact with other units of code. But this intent of programmers is typically implicit and undocumented, goes beyond considerations of functional correctness, and may depend on trust assumptions that programmers make. At present, neither programming languages nor development environments offer a means of articulating such intent in a manner that could be used for controlling whether software executions meet such intentions and their associated expectations. We here study how extant research on trust can inform approaches to articulating programmers' intent so that it may help with creating trust evidence for more trustworthy interaction of software units.

We first describe a known model for expressing the mechanics of trust in transactions between two parties. Then we sketch a possible technical approach that allows programmers to capture intent in the form of expectations about method invocations. We then demonstrate how this approach may generate quantitative trust evidence that can form the basis for deciding whether methods should be executed. Finally, we discuss to what extent this technical approach reflects the model of trust mechanics and identify future work in this space.

1 Trust Mechanics

We begin with a discussion of the mechanics of trust in transactions by recalling research in that area from the social sciences and the usability of security.

Trust versus Assurance. In the real world, individuals and organizations cooperate to achieve mutual benefits. But this only works out if both parties fulfill their side of the bargain. As Flechais et al. [2] point out, we can carry out a risk assessment for a specific cooperation, and deploy mechanisms that make it hard for our transaction partner to cheat us; such an assurance strategy does, however, come at a cost, which reduces the benefits we reap from the cooperation.

The second strategy allows cooperation partners to save the cost of assurance, by trusting each other to behave as expected. Mayer et al. [8] define trust as: *"... the willingness to be vulnerable, based on positive expectation about the behavior of others."* The idea of *"willingness to be vulnerable"* is the antithesis of the traditional security perspective, whose raison d'être is the creation of mechanisms to prevent exploitation of vulnerabilities.

But even the security community has become aware of the cost of assurance, which sometimes consumes the benefits that can be reaped from the cooperation. Consideration of the economics of security has become a thriving sub-discipline, with the Workshop on Economics of Security (WEIS), founded by Jean Camp and Ross Anderson, now in its 11th year. Trust provides economic benefit to cooperation partners – provided neither of them cheats. If there are too many incidents of cheating, we need to add security mechanisms that make cheating harder, but doing so comes at a cost.

Leading security expert Bruce Schneier calls these cheater *"Liars and Outliers"* in his latest book [12]. In a shift of perspective that must be a shock to many of his devoted audience of security professionals, he argues that – rather than develop costly mechanisms to deter and prevent cheating – we must shift our focus to designing systems that foster and incentivize trustworthy behavior, so modern society can reap the benefits that come with cooperating and trust-based interactions. Without cooperation and trust, individuals and organizations, economies and societies cannot thrive [4].

Disembeddedness. The introduction of modern technology has significant implications for trust. Giddens [3] was the first to point out that the knowledge on who we can trust, when, under which circumstances (something we learn from our parents and other sources of social authority, and though experience) – is very much embedded in a particular space-time context.

Modern technology has enabled collaborations that are *disembedded* from space and time – more and more interactions are now taking place without the transaction parties ever meeting in face-to-face, in the same place. Disembedding makes cheating easier: in a shop, it is unlikely that the shopkeeper will take your money, and then not hand over the goods you just paid for; with online transactions, you don't know for days if the seller in Transaktistan will send the artisan set of nesting dolls you just paid for. And you won't know for weeks, months or maybe even years whether the seller decided to sell your credit card information to someone else.

But our mental models of trust are still very much dominated by the embedded transactions. Kirlappos et al. [5], for instance, found that even experienced Internet users trust websites based on familiarity (*"looks like one I've used before"*), and based on the apparent presence of links to social networking sites and charities – and are unaware of how easily these trust signs can be forged in the online environment. The *Mechanics of Trust* framework by Riegelsberger et al. [11] examines trust signaling online.

Trust-Mediated Interactions. We now consider interactions between two parties, where the interaction is mediated by trust. The two parties are the trustor (who exposes a vulnerability to the other party, in hope for gaining a benefit from this) and the trustee (who may or may not provide such a benefit to the trustor). A schematic of such a transaction is depicted in Figure 1.

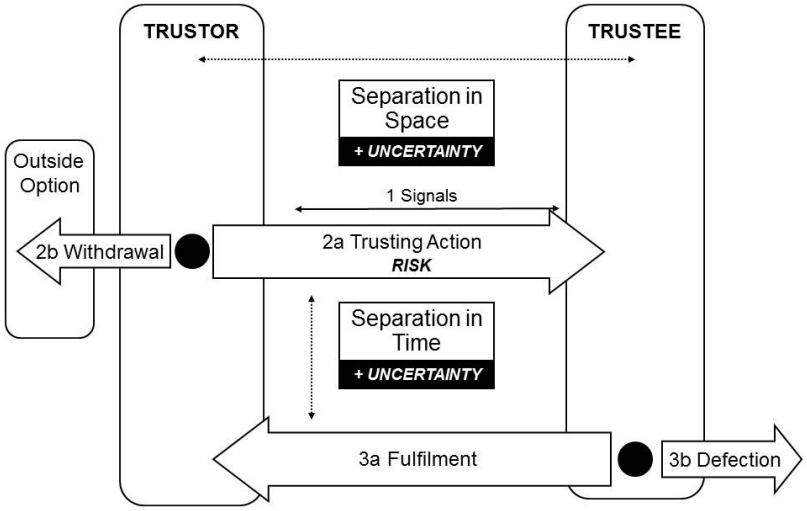

Fig. 1. Trust-Mediated Interaction between Trustor and Trustee who are separated in space and in time, figure reproduced from [11]

During the first phase of a Trust-Mediated Interaction, the trustor and the trustee exchange signals, which are used to assess each other's *ability* and *motivation* to engage in a successful transaction. After the initial signal exchange the trustor has the option to either proceed to the trusting actions (2a), which renders them vulnerable to the trustee's behavior (2b), or withdraw, in which case the transaction ends.

The decision is not solely based on the signal exchange though: trustor's risk propensity, perception of exchanged signals and other external factors (e.g. existence of easy withdrawal) also affect the choice between trusting action and withdrawal. After the trusting action the trustee has full control over the situation and they can choose to either fulfill (3a), by behaving in the way the trustor expects, or defect, having already obtained the benefits of the trustor's trust (e.g. monetary sums or financial details).

The advantageous position in which the trustee is after the trusting action means that fulfillment will occur when the trustee realizes some gain from it; otherwise trustees are better off defecting and reaping the benefits. As a result, fulfillment – which successfully ends the interaction – will occur only when the trustee has *both* the ability and motivation to fulfill. And therefore the trustor needs to be able to 'read' the trustee's trust signals to determine both ability and motivation are present.

Trust signals and trust symbols. In online transactions, the technology is the channel through which trust signaling occurs. Prior to the occurrence of a trusting action, both parties transmit signals, resulting in a perceived level of trustworthiness of each other. By reading those signals, the trustor forms expectations on the behavior of the trustee. There are two types of trust signals: symbols and symptoms

1. *Trust symbols* have arbitrarily assigned meaning, and were designed specifically to signal the presence of trust-warranting properties. Trust seals, for instance, signify that a seller is a member of a scheme, and promises to abide by its rules. There are many problems with trust seals [5]: they can be easily forged, and customers over-interpret the level of protection they offer. Reputation mechanisms (such as the 'star' rating on ebay) can be subverted through the creation of multiple identities, shill bidding, and 'cashing in' established reputations. For instance, a fraudster can buy an existing online shop with a good reputation, offer attractive prices at a busy time of year, and then just take payment without shipping the goods. By the time their actions are discovered, the damage is done.
2. *Trust symptoms*, on the other hand, are signals given off as natural by-products of honest transaction partners going about their business. So a business that takes a certain level of payments for goods has to ship a certain amount of goods. There are a limited number of shipping agents they can use, and transmitting information of their shipping activity incurs no extra cost to an honest seller. But for a fraudster, creating fake shipments would be a considerable cost. Another example of a trust symptom, would be that a commercial website has an easily usable and well thought out user interface with a coherent design look; this creates the impression of professionalism that may be conceived as a symptom of trust.

2 Trust-mediated interaction of units of code

The trust mechanics described above is based on the notion of a transaction: will it come to a trusted action or not? And this mechanics involves two parties: the trustor who may take the risk of committing to the trusted action, and the trustee who may or may not fulfill the expectations or obligations inherent in the trusted action. We now examine to what degree this model of trust-mediated interaction may be of use in articulating and enforcing programmers' expectations of code

and its interaction with other units of code. Let us here consider the scenario where a unit of code is a method (also called a *procedure* or a *function* in various programming languages).

Methods may have parameters and – when invoked by another method – instantiate such parameters with concrete values, execute a method body within these instantiations, and may return a value as result (or may change state as a side effect). An example method name `withdraw` with its output type and list of parameters is `int withdraw(Account acc, int amt)`. Its method body (not shown here) aims to withdraw the amount `amt` (an integer) from the account `acc` and return the new account balance (also an integer).

Programmers will often articulate intent that concerns functional correctness, and they can do this in program documentation, in `assert` statements that check whether Boolean predicates hold at specific program points, or through other executable program annotations (e.g. as in the Java modeling language [7]). For method `withdraw`, e.g., a programmer might stipulate that `amt` has to be non-negative (otherwise, one could potentially withdraw money and increase the account balance at the same time) and greater than the account balance, etc..

But there are other expectations that the programmer is likely to have that cannot be articulated through the aforementioned means. For example, a programmer might know that method `withdraw` will only ever be called (i.e. invoked) by another method `authenticatedWithdraw`. Yet this knowledge is at best expressed in program documentation and so cannot be harvested for informing and potentially controlling the execution of such units of code.

A programmer may further expect that certain tasks within a workflow of code may be completed more frequently than others. For example, that over the lifespan of a social network user account there are at least, say, three times as many successful login attempts than requests to recover or change a password.

Another expectation that programmers will have is about the range of data values for program identifiers. For example, a programmer may expect that some input parameter `x` is what is known as a *safe* prime. One could specify such a requirement as a precondition for input: that the value of `x` is a prime number of form $2q + 1$ where q is itself a prime. But programmers might still want to execute a method if the value of `x` is not a safe prime but at least a prime. Yet, this deviation from expectations may decrease the trust in running the method.

Conceptually, we may think of a programmer as collecting expectations on normal method execution. And a run-time system could then determine which of these expectations are true. And this set of observably met expectations would then, ideally *compositionally*, compute the overall trust evidence for executing the method body. We note that it is far from obvious to answer how such evidence should be composed so that its composition be meaningful to the programmer and consistent with the 'sum' of his or her expectations on program behavior. One challenge here is that expectations may be quantitative (e.g. in the login example above) or qualitative (e.g. in the method invocation example above).

Relating trust evidence annotations to trust mechanics. We now discuss how such use of annotations for trust evidence generation and enforcement

relates to the trust mechanics depicted in Figure 1. We begin by identifying the roles of trustor and trustee.

The trustor seems to be the programmer, whose annotations make it possible to assess the risk of committing to a trusted action – which in this case would be the execution of a method body. We may equally think of the annotated method as the trustor who ponders whether it should be executed. And if we were to implement these annotations by integrating their evaluation within a run-time environment, we may think of the run-time environment (e.g. a modification of the Java Virtual Machine) as the trustor.

In this setting, the trustee is the method calling the annotated method. Interestingly, it would also make sense to think of calling methods as trustors since a called method may return values that calling methods further process and so may need to trust. The above approach is consistent with embracing this view as well. A calling method `foo1` may itself have intent of using another method `foo`, and this intent may lead to annotations within `foo1` that may be used to evaluate whether or not to call `foo`. And the results of `foo`, if called, may then be used to judge whether the interaction met the intent of `foo1`.

Returning to the view where the annotated method is the trustor, its decision to withdraw (2b in Figure 1) comes about when its annotated policy computes as decision value `deny`. If that value computes to `grant` instead, it will take the risk of performing the trusted action that executes its method body.

As for the signals (1 in that figure), the trustor takes the predicates expressed in expectation blocks as trust signals that, when true, trigger the generation of quantitative trust evidence – through the declaration of trust evidence scores, composition operators, and default composition values.

Separation in space, in time, and uncertainty are also relevant in this approach. Calling methods may stem from different machines, different domains, etc.. A process-based program model such as Actor languages [1] would definitely exhibit separation in time as well, due to their asynchronous communication.

Trust evidence generation. We now sketch what sort of annotation support might be able to express programmers' expectations, and the composition of the trust evidence that each expectation would generate.

First, let us consider ways of articulating the trust we may have in particular callers or groups of callers for a given method. We may write an annotation block and place it in front of the method declaration, as in

```
@expect[max] default 0.1 {
  if (calledBy foo1) setTrustEvidenceTo 0.9;
  if (calledBy foo2) setTrustEvidenceTo 0.3;
  if (sameDomain(@caller)) setTrustEvidenceTo 0.8;
}
method foo(...) { ... }  // body of method foo(...)
```

This defines an expectation block with three statements that each capture expectations on executing method `foo`. The first statement specifies that the trust evidence for executing method `foo` is 0.9 if the caller of that method is method

`foo1`. In contrast, if the caller is method `foo2` that trust evidence is only 0.3. Finally, the predicate `sameDomain(@caller)` may express that the method that invokes `foo` is from the same system domain as method `foo`, in which case the trust evidence would be 0.8.

The annotation `[max]` directs to compute the maximal such trust evidence, whereas `default 0.1` says that the composed trust evidence is 0.1 if all three predicates are false (i.e. when the method that calls `foo` is not from the same domain and different from `foo1` and `foo2`). For example, if the method that calls `foo` is `foo1` and if `foo1` happens to be from the same system domain as `foo`, the composed trust evidence would be 0.9 since `max` is used for composition.

The composition operator `max` takes an *optimistic* view of trust evidence: take the most trustworthy evidence as basis for assessing the risk of committing to a trusted action (here: executing the method body). Of course, we may also want to write expectation blocks that are composed through `min` and so take a *pessimistic* view on trust evidence, by considering the least trustworthy evidence as a basis of decision making. Alternatively, a block `@expect[+] {...}` may additively *accumulate* trust evidence (with composition `+`): the sum of all the evidence becomes the baseline for assessing risk of committing to a trusted action.

And we may even conceive that programmers may want to specify several such blocks with different composition operators, and then also specify how the composed evidence of each block should be composed to an overall evidence. For example, we may consider the minimum of the evidence computed from a pessimistic, an optimistic, and an additive expectation block to compute the trust for executing the method body in question.

Usability issues. Whatever an approach to capturing programmers' intent of method interaction may be, it needs to be simple enough so that programmers can reliably capture their expectations in it. It also needs to be expressive enough so that key expectations can be formulated in it. This suggests that such a language should be *extensible* as we cannot anticipate the needs and intents of programmers for general code development.

Furthermore, a critical issue is that the semantics of *evidence composition* is both natural and intuitive enough for programmers and also consistent with the intent that programmers had in mind. We believe that techniques from program analysis [10] can be transferred to such annotation support in order to generate diagnostic information that can help programmers to validate that their `@expect` annotations are consistent with their actual intent. To illustrate this on a very simple example, we may edit the above expectation block to

```
@expect[max] default 1.0 {
  if (calledBy foo1) setTrustEvidenceTo 0.9;
  if (calledBy foo2) setTrustEvidenceTo 0.3;
  if (sameDomain(@caller)) setTrustEvidenceTo 0.8;
}
method foo(...) { ... }  // body of method foo(...)
```

which changes the default value to 1.0, a numerical indication of complete trust. This seems inconsistent in the following sense: the block formulates three ob-

servables that, when true, might constitute trust evidence. But if all observables are false, this results in a higher trust evidence 1.0 than the one we would assign, say, when `foo1` is the caller of this method.

Put in another way, this seems to trust callers that are not in the same domain and different from the two named methods in the block more than any other callers! In particular, this would apply to some unknown method from another system domain. One sensible integrity constraint for the default value for `max` composition may thus be that it be equal to the minimal trust evidence value in that expectation block (i.e. 0.3 in this case). Semantically, this would make the default value redundant, but having it explicitly in the syntax may help with better comprehending these annotations and their meaning.

Trust evidence enforcement. We do not suggest that this approach to trust evidence generation is the only feasible one or one that can work *as is*. But whatever programmatic means are used for generating quantitative trust evidence of method invocations, it raises the question of how to enforce such evidence.

We suggest to use simple policies for articulating contextual and other circumstances under which a method should be executed. Let the programming language have a reserved keyword `localTrust` in which we store the composed trust evidence, computed as above. Then a very simple policy may be of form

```
@policy{
  grant if (localTrust > threshold)
  deny otherwise
}
```

where `threshold` is some value chosen by the programmer, e.g., 0.6. In that case, the method would be executed if the composed trust evidence is greater than 0.6; and this execution would be blocked if this evidence is ≤ 0.6.

It is attractive to have such a simple and uniform policy pattern. But, in a way, such simplicity just shifts the complexity into understanding the circumstances in which the value of `localTrust` exceeds 0.6, say. Ideally, we would want static analysis support that can inform us about the scenarios in which there is sufficient trust evidence. And such diagnostic information may lead the programmer to revising his or her expectation specifications or it may confirm that these specifications meet the programmer's intent.

Some applications may require more complex policies. For example, qualitative evidence is best expressed at the policy level itself, e.g., using rule formats as familiar from the OASIS standard XACML [9]. And so its combination with quantitative evidence may result in other policy composition patterns such as

```
deny if (calledBy evil)
grant if (localTrust > threshold)
deny otherwise
```

which lists rules in a priority ordering so that method execution is certainly blocked when the call comes from a method `evil` and where otherwise the policy evaluates as in the above composition pattern.

We note that policy decisions may not only result in either blocking or allowing a method execution; a policy decision may for example report an inconsistency which may lead to an execution of that method with modified input. Although this has ramifications for language design and usability, we do not present such a more complex enforcement mechanism in this paper.

3 Discussion

Looking at the trustee's options of Fulfillment (3a in the figure) and Defection (3b in the figure), we interpret these terms from the perspective of the trustor (as intended in this trust mechanics). Fulfillment here seems to mean that when the annotated method body is actually executed it turns out that this decision to run that code was beneficial. And Defection seems to suggest that granting that execution leads to some undesirable outcome. Our vague language is not accidental: what 'beneficial' and 'undesirable' mean depends on circumstances.

Another issue is whether these annotations can also model the possible evolution from trust in signals and systems to a reliance on behavior based on past positive interactions. One idea might here be to allow for the update of trust evidence scores in annotations. And there is the issue of how to derive initial trust scores, which may be informed by a programmers' risk posture.

The last two points asked what Fulfillment and Defection might mean and whether trust scores may evolve based on the interpretation of these actions performed by the trustee. We think that these questions are intimately related to the notion of *intent* that the programmer would have when developping or using units of code.

To illustrate this relation on a simple example, let a programmer develop a new sorting algorithm `sort` that delegates core sorting work to an auxilliary function `sort0`. The programmer's intent behind this division of labor is that `sort0` should only be called by `sort` and not by any other method. So in this case, the intent and the possible annotation (which may assign trust evidence 1 when the call comes from `sort` and trust evidence 0 otherwise) are at the same explanatory level.

A more complex example would be when a programmer has the intention that a unit of code only be used with other units of code, *provided* that interaction is strictly about business. So this may rule out interactions with third-party applications, access of certain web pages, and use of an editor if the edited text is not work related, etc. As this suggests, it will be much harder to refine such abstract intentions into annotations that are close enough to the application layer in order to be enforceable. The problem of refining policies to cross semantic layers has been studied in the context of usage control [6], and it would be interested to see whether these techniques may be applicable here.

But we note that a first research problem here is to actually come up with usable languages in which high-level intentions could be captured, before we may consider how to refine such specifications.

4 Conclusions

We have identified that there is currently no adequate support for expressing expectations that programmers may have on the interactions of units of code. Then we described a technical approach that provides a partial solution to this problem and that seems to conform with a standard model for trust mechanics in trust-mediated interactions. We also identified an explanatory gap between *intent* and enforceable programm annotations that needs to be closed so that high-level considerations can be reflected and enforced in low-level programming.

Acknowledgments

We acknowledge the kind support of Intel® Corporation: the first two authors are funded within the Intel® Corporation *Trust Evidence* project; and the last author is funded for a grant *Teaching Security Through Serious Games*.

References

1. Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems.* MIT Press series in artificial intelligence. MIT Press, 1990.
2. Ivan Flechais, Jens Riegelsberger, and M. Angela Sasse. Divide and conquer: the role of trust and assurance in the design of secure socio-technical systems. In *Proceedings of the 2005 workshop on New security paradigms*, NSPW '05, pages 33–41, New York, NY, USA, 2005. ACM.
3. Anthony Giddens. *The Consequences of Modernity.* Cambridge: Polity, 1990.
4. Charles Handy. Trust and the virtual organization. *Harvard Business Review*, 73(3):40–50, May 1995.
5. Iacovos Kirlappos, Martina Angela Sasse, and Nigel Harvey. Why trust seals don't work: A study of user perceptions and behavior. In *TRUST*, pages 308–324, 2012.
6. Prachi Kumari and Alexander Pretschner. Model-based usage control policy derivation. In *Engineering Secure Software and Systems*, pages 58–74, 2013.
7. Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accomodates both runtime assertion checking and formal verification. In *First International Symposium on Formal Methods for Components and Objects*, pages 262–284, 2002.
8. R. Mayer, J. Davis, and F. D. Schoorman. An integrative model of organizational trust. *Academy of Management Review*, 20(3):709–734, 1995.
9. Tim Moses. *eXtensible Access Control Markup Language (XACML) Version 2.0.* OASIS Standards Committee, February 2005.
10. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis (2. corr. print).* Springer, 2005.
11. Jens Riegelsberger, Martina Angela Sasse, and John D. McCarthy. The mechanics of trust: A framework for research and design. *Int. J. Hum.-Comput. Stud.*, 62(3):381–422, 2005.
12. Bruce Schneier. *Lairs and Outliers: Enabling the Trust and Society Needs to Thrive.* John Wiley & Sons, 2012.