Generative Usability: Security and User Centered Design beyond the Appliance

Luke Church
University of Cambridge
Computer Laboratory
Cambridge, UK, CB3 0FD
luke@church.name

Alma Whitten
Google
76 Buckingham Palace Road
London, UK, SW1 9TQ
alma@google.com

ABSTRACT

In this position paper we consider the ways in which users can be given control over technology and information, considering the spectrum of design possibilities from 'generative component' solutions, to 'appliance' solutions. We show how security concerns and the processes of user centered design tend to encourage a migration towards the appliance end of the spectrum and then describe problems that arise from this. We then suggest an alternative route towards allowing users more direct control over their information via end user programming, discuss some of the challenges in doing so and how they might be overcome and conclude with a suggestion of a practical first step that system designers might consider.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – User-centered design, Graphical user interfaces (GUI); D.4.6 [**Operating Systems**]: Security and Protection – *Access Controls*; K.4.1 [**Computers and Society**]: Public Policy Issues - *Privacy*

General Terms

Design, Security, Human Factors.

Keywords

Security Usability, HCI-SEC, Appliance, End User Programming, Meaningful choices

1. INTRODUCTION

What is the purpose of Computer Security? Arguably, it is about giving people control over computers and information. It shares this goal with Human Computer Interaction (HCI), but has often approached the problems from a different direction. In this paper we discuss a challenge that both disciplines are facing: what is the best way of offering users such control, though appliancisation, or further flexibility?

We examine the pressures towards appliancisation in both disciplines and consider some of the implications of these decisions, which as Zittrain puts it, may shape 'The Future of the Internet' [28].

Expanding on Zittrain's analysis, we then consider some of the challenges that need to be addressed in order to support 'generativity'. We conclude by synthesizing some of the lessons from HCI and Security and discussing some directions that seem promising.

2. THE STATUS QUO – HOW ARE WE DOING?

First, let's consider the state of technology. The usability of computers has improved dramatically. This is evidenced both by their commercial success and widescale deployment to users with little formal training, as well as in personal correspondence with Clayton Lewis, co-author of a seminal early paper on usability [12]**.**

A lot of the gains in usability appear to have come from adopting a direct manipulation style of user interface [25]; with operations being performed on visual representations of objects, rather than by formulating abstract commands to be applied later.

Consider for example, a standard desktop file system. In the days in which Gould and Lewis wrote their paper, moving a text file from one directory to another would have involved typing an instruction such as '`mv nspw.txt /home/old/paper`' – now, it would involve dragging an iconic representation of the file onto an iconic representation of the folder. For users who don't want to do large numbers of complex operations, this seems like progress.

But, if we consider the challenge of setting access permissions on the same file, it's a different story. Where the old way of doing this involved using something like the `chmod` command, the UI version is only cosmetically different, with tick-box representations of the various options – but with no substantial fix for the underlying challenge.

Security configuration interfaces are like this, Whitten [27] described it as the *Abstraction property:*

"Computer security management often involves security policies, which are systems of abstract rules ... The creation and management of such rules is an activity which programmers take for granted, but which may be alien and unintuitive to many members of the wider user population. User interface design for security will need to take this into account"

Work in the psychology of programming has characterized programming as the loss of direct manipulation [3] - showing that end users begin to program either when they create operations to be performed in their absence or operations to be performed repeatedly over lots of objects. Security configuration, almost by definition, is doing exactly this. The abstraction property can therefore be strengthened from an observation, to an inevitable consequence of the nature of security.

A similar conclusion applies to the *Lack of feedback property*, also by Whitten:

"The need to prevent dangerous errors makes it imperative to provide good feedback to the user, but providing good feedback for security management is a difficult problem. ...., the correct security configuration is the one which does what the user "really wants", and since only the user knows what that is, it is hard for security software to perform much useful error checking."

Again considering our example of the file system; the feedback from moving the file is relatively good, especially in the graphical version. It now appears contained within the new folder, as opposed to the old one. For the security configuration system, as suggested above, there is little that can be easily done other than displaying what the user has chosen and allowing them to check its correctness.

This is another characteristic of programming systems – because they are rules to be employed over future information, their state space is usually too large to directly visualize and 'errors' look, at best, like unusual configurations. So designing feedback now shifts from being about designing state visualizations (like which files are in which folders) to designing tools to help users reason about their system, again undoing direct manipulation and producing another layer of abstracted behaviors. If anything, the presence of a malicious advisory in security amplifies this challenge; the likelihood, and impact of, the user experiencing problems due to an unknown state is higher than when they're only dealing with bugs triggered by random chance.

This leaves us in a difficult position when trying to build security systems that are usable. We've got most of the usability in computer systems by removing the elements that were programming – we now either have to begin adding them back in, or we have to defer the decisions to experts and embed these decisions into the systems we are trying to build – appliancising the security behavior of the system.

It is unsurprising that many engineers, when trying to give users control over their technology and information, have opted for the later solution. We'll return to the former in Section 5. But for now, let's expand a little on what we mean by the appliance model of security.

3. TOWARDS APPLIANCES?

We join Zittrain [28] in suggesting that there is a continuum of user freedom in technology, between 'appliances' and 'generative technology'.

Let's start with an example from domestic technology. Toasters are an example of appliances, they're good at cooking pre-sliced bread, but that's pretty much all they're good at. Ovens on the other hand are at the generative end, they aren't so easy to use to make toast, but are far more flexible – and can be used to cook things that their inventors hadn't thought of.

With computers, things get a little more subtle, but the continuum is still there. A PC is a long way towards the generative end – it can be reprogrammed by its owner in pretty substantial ways, a Tivo on the otherhand is an appliance. It serves only a single purpose and cannot be substantially reprogrammed[1] by its user.

Zittrain argues that computing technology is at a critical point; the flexibility of generative PCs has resulted in a proliferation of instabilities and 'badware'. Out of a frustration at the insecurities of their generative computers, users may turn to locked-down appliances, requesting e.g. single purpose phones [17]. In such devices, security technologies start to serve a dual purpose – protecting the end users' information, and more problematically, 'protecting' the device from the end user. Zittrain was talking primarily about the flexibility of devices, but the same debate applies equally to software systems. Unable to offer meaningful security policy choices, we are seeing increasing numbers of systems which are deployed with a small number of preset policies from which the user may choose. This is the 'appliancisation' of the security model of the system. We suggest that this approach has serious problems, and we should not be too quick to adopt it, without giving its harder alternatives a fair trial.

First, let's consider another pressure towards the appliancisation, from a possibly surprising source – User Centered Design.

3.1 User Centered Design and the Appliance

User Centered Design (UCD) is a collection of design philosophies and methodologies focused around building technology to serve users. The paper mentioned earlier by Gould and Lewis [12] which is considered to be one of the canonical works [21] has as a central aspect the 'early focus on users and tasks'.

UCD has been interpreted and incorporated into a large number of frameworks and processes. In order to understand how it influences the drive towards to appliancisation, let's consider the version discussed in Jenson's excellent practical guide to UCD in industry [16].

Jenson stresses four blindnesses that can hamper the product design, the two that concern us here are: *user blindness* – making incorrect assumptions about who the user is, and *feature blindness* – products becoming awash with sophisticated features.

Jenson recommends the use of *scenarios* and *personas* to address these blindnesses. These are approaches which seek to produce representations of concrete instances of tasks, contexts and people around which to determine functionality. This helps avoid unduly abstracting away from users, which we have previously argued [4] is a major cause of usability problems in modern software.

Greatly over-simplified UCD could be seen as saying that technology design should be done by finding out who your users are and what they want to do, and building technology to do that. The difficulty is that this applies pressure towards the appliance model of technology. In order to see how, let's look at a design maneuver that is recommended by Jenson, and then see how it often occurs in complex cases – such as security configuration.

One of the solutions that Jenson advocates to feature blindness is "Make the Easy, Easy and the Hard, Hard". This common-sense recommendation suggests that commonly used features should be

[1] We need to be careful not to confuse terminology here. There are degrees of 'reprogrammability' – a Tivo can be reprogrammed to record a different program, but can't be reappropriated to support e.g. mashing up of different TV channels

promoted to the front of the user interface, whilst advanced features should be hidden. The rationale being that the users of the advanced features are precisely the user community who can best cope with the additional operations.

Like much of the UCD agenda, it would seem perverse to find much fault in this suggestion. Placing little used, complex features, deeper inside menu hierarchies than commonly used ones seems sensible – especially if one believes as Jenson and Maeda [16, 20] do, that there is an economy of usability – one can't make something easier without making something else harder. This makes some sense when talking about screen real-estate, but it also occurs in a more complex form in the design of programming-like systems, of which we argued that security configuration is one.

A common problem in configuration systems is *viscosity* (resistance to change – frequently caused by having to perform the same operation multiple times). This is tedious and annoys computer scientists[2]. A typical design maneuver to address this is to introduce an *abstraction*. [14]. This allows a collection of entities to be named, and changed simultaneously – either by the programmer editing them, or by adopting the abstraction provided by an expert. However, adding an abstraction to a system has side-effects; Firstly, it generally increases *premature commitment* (the degree to which users have to make decisions ahead of time). The decision as to what the appropriate abstraction is typically has to be made a long time before it's used, this is especially the case when it's supplied by an expert. Secondly, it reduces viscosity, if the abstraction aligns to what you want to do (in our terminology if it is *role expressive*). If the abstraction doesn't align, then at best viscosity remains the same, more usually, it actually increases, usually because the designers of the UI have followed Jenson's principle – they've made what they believe is the common operation, the one that aligns with the abstraction, easier, so they can bury the alternative, manual version, deeper in the menu hierarchy.

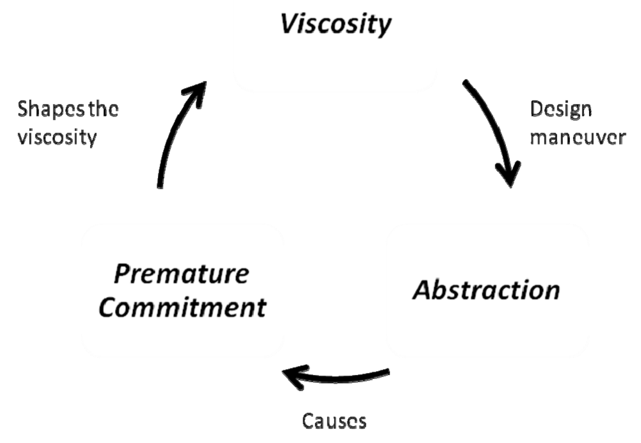

Figure 1 - The viscosity shaping cycle

[2] E.g. Terrence Parr's motto: "Why program by hand in five days what you spend five years of your life automating" - http://www.parr.us/terence/index.html

This effect, *viscosity shaping* (summarized in Figure 1) results in a *normative force* as to how the technology should be used – users are strongly rewarded for taking the 'easy route' that fits with the abstractions. If this normativity is taken to its limit, an appliance is the result; this makes the easy, easy, but the hard, impossible.

3.2 Summarizing the pressures of appliances

At this point, it's worth standing back and reviewing what's happened here. Our original goal was to give users control over technology and information. The way the appliance achieves this is to have an expert decide, ahead of time, what the product can and cannot do.

This is a slightly odd conclusion – that we give people control by having someone else define what they can do. However, every step along the way was well intentioned, to protect the user from the side-effects of insecurity and because it's easier to achieve usability by embedding pre-defined normative abstractions, than by building systems whereby users can painlessly express their own.

So the pressures of both security and user centered design all seem to incline towards appliances.

4. PROBLEMS WITH SECURITY APPLIANCES

In Section 3 we reviewed the pressure that security and user centered design cause towards appliancising, embedding security behaviors. In this section we look at the problems of this approach.

The difficulty is that, as suggested above, it's a questionable form of control. This creates problems with the user experience – at best the user is presented with a number of options but without understanding the decisions embedded within them, there is little they can do beyond hoping that the mental models they have of what is provided is accurate. A study on users' assumptions about the privacy behaviors of websites that have a privacy policy is not encouraging in this regard with many users assuming that just the presence of a policy meant their data was strongly protected. [15]**.** If users' model of the presence of such policies is so problematic, we certainly should not assume that their models as to what the policies do are accurate.

The usability problem here is fairly inescapable – how can a user make meaningful choices about the selections of experts? One option is blind faith, that the users must just trust whatever the expert has given them, but this is hardly a meaningful choice and if our goal is to give users control, a 'solution' that we should be reluctant to adopt.

Another option is for the expert to provide summaries of their policies, and the user can choose between the policies on basis of the summaries. But as both authors have pointed out independently, security is one of the places where this kind of summarizing abstraction doesn't seem to work well [27, 6].

A final option is for the users to understand the policies provided, and choose between them in full knowledge. However, if they are capable of doing this, why not also allow them to edit the policies?

We suggest that this is important as there is another aspect to the question – *that a user being able to intentionally express their security wishes creates a qualitatively different experience than choosing from a selection of opaque options provided by experts*. We suggest that supporting this experiential different might be crucial for building systems which allow the users to trust that their wishes will be respected, rather than the current cynicism over security/privacy behaviors – one of the challenges Zittrain argues needed to be addressed.

There are also systemic issues with appliancisation; as suggested above it represents a shift away from our core suggested goal for security and HCI, of giving users control over technology, to one of giving technologists control. There is always going to be some need for negotiation between users and producers, but this is a particularly worrying form of 'negotiation'. In many cases, when we appliancise security policies, we dramatically shift the power balance towards the technologists, essentially removing the ability for technically minded end users to use their skills to contribute to the conversation.

This places an impossible requirement on technologists: we must predict all the ways in which people might use the technology we create, and create security policies that support all the positive aspects. Would we security professionals, in January 2001, honestly have predicted that a system that allowed arbitrary users to write information to a website, and that using a blend of social and technical mechanisms to prevent 'abuse' was a good idea? Yet, Wikipedia succeeds. In an appliance model, such as the walled gardens of AOL and Compuserve, the administrators would have had to decide in advance that this was something they wished to support, but our ability to predict technology in this way has historically been notoriously poor.

Market forces don't provide much of a solution to this problem either. The tendency of networked systems towards monopolies has become almost a truism of modern economics. [24] Even if there was a range of options to choose from, for the same reason that users that users struggle to understand the configuration options provided by experts it is something of a market for lemons, where the buyers and sellers have asymmetric information about the product, and the buyers therefore struggle to make accurately informed choices.

Another systemic issue is the incentives shift that occurs when control is taken away from the users and given to the technologists associated with companies. Zittrain goes into considerable detail discussing the threats of this which range from 'perfect enforcement' – using the security policies delivered by experts to ensure that illegal things can't be done, rather than punishing them after the fact, to the loss of 'tolerated uses' – where technically illegal but not realistically enforceable practices grow into accepted uses of the technology and 'surveillance' – that companies tend to use appliances to accumulate data about their users, even when they have nothing to do with the vendor, and that this turns them into honey pots for judicial authorities.

All these concerns with appliancing security models are particularly serious in domains for which the social norms are still being negotiated. Privacy is one of these. We do not yet know how much information individuals will be happy disclosing in order to improve advertising results – whilst we have to guess in order to build technology, the appliance model of security policy would have us assume that those guesses were correct, rather than offering users the ability to suggest changes.

Privacy is uniquely susceptible to this problem, as we don't even have good ways of gathering people's desires in indirect manners. In [1] Acquisti and Grossklags shows that users' decisions change radically dependent on context. If we adopt the appliance model, we will have to understand all these behaviors and embed them into our technologies. This goal currently seems somewhat elusive.

Finally, there are problems as to how well behaviors generalize. Does an 18 year old European university student have the same privacy desires as a 75 year old Asian rural pensioner? How will we cope with this diversity if we are to make the choices ahead of time? How will we cope with the diversity of different policies that we'll need to offer – ultimately the thing that we appliancised to avoid having to do in the first place?

To summarize: we are suggesting that there a number of serious problems with appliance models for security policy, and that *we should not attempt to predict all of the complexity of our users' wishes ahead of time, but should instead design tools to allow them to meaningfully express their wishes.*

5. THE CHALLENGES OF MEANINGFUL CONTROL

If we are to attempt to design security technologies without resorting to appliancisation, there are a number of challenges to be met. We outline these challenges here and then look at some promising pointers in each direction.

The challenges spread right across the disciplines of Security and HCI – from tactical problems of how to design user interface mechanisms, through to strategic challenges of how to have spectrum discussions about security. The problems are hard, but we don't need perfect answers in order to make progress.

5.1 Mechanisms

Let's start at the mechanism scale. We have argued that security configuration is an act of programming – but most of the current user interfaces are using design techniques that were specifically developed to avoid their users having to do programming. Take for example Facebook's privacy configuration UI, at the time of writing it contained 61 drop options spread across 7 screens. There are a number of problems with doing this that limit the use of interfaces for providing meaningful control. The most obvious are *viscosity*, and *diffuseness*.

In order to assess, and improve such interfaces we need a way of talking about the *usability of programming mechanisms*. Here at least, there is some progress – viscosity and diffuseness are both parts of the Cognitive Dimensions of Notations Framework [14], a structural tool for discussing the usability of notations that are used for programming. As a framework it has some problems, for example it is comparatively difficult to learn, a problem that we are actively addressing. More problematically for our purposes, it is not sufficient to talk about the usability of programming mechanisms – in order to give meaningful control, we must also consider the *user experience of programming*.

Consider the case of discovering that someone gained access to some information on Facebook that you didn't expect them to be able to. It is not sufficient to be able to change the appropriate settings to ensure that such access couldn't take place, it is also crucial that when modifying your privacy settings you have some confidence in the results of your changes. What creates such feelings of confidence? What inhibits them? At what level of flexibility does it become more confidence inspiring to use programming-like techniques? How can we ensure that these feelings are grounded in actual security properties, so users may safely use them as a heuristic?

Here the progress is more modest. Results in other areas of psychology of programming and visual language design give some good pointers to start with, e.g. the ability to query an IDE as to why a specific behavior occurs [18], and work in the factors that affect user confidence in spreadsheets [2]. An interesting case to consider here is programming by example – where the system attempts to infer abstractions automatically from a series of direct manipulations provided by the user. We have previously suggested that this might be an interesting approach to addressing end user programming for security [8], however whether it possible to build such systems in a way that inspires sufficient confidence that users would feel happy configuring their security systems this way remains an open question. Exploring this seems a promising route to understanding the experiential, rather than purely computational effects of program construction.

We have some tentative progress in expanding these areas of research into a framework for discussing the user experience of programming but much work, both theoretical and empirical, is needed.

5.2 Strategies

Moving away slightly from the mechanisms, there are challenges in supporting the higher level strategies that end users may wish to use to go about configuring security systems. It is often considered in software engineering to be a good idea to think hard about the design of a system, before beginning to build it. Languages have tools, like type systems, built into them to support this behavior. These tools bring *premature commitment* and *useful awkwardness* – the act of thinking hard about the system in a particular way is useful in finding some types of problems.

However, this strategy is not the only possible one. An alternative is *exploratory design*, supporting the user in an activity whereby they fluidly change the data structures and behavioral rules, consider the behavior of the system they have created and rapidly iterate.

This is some evidence that both professional programmers [9] and end user programmers [19, 26] employ a range of different strategies. However, we know very little about what strategies support a positive user experience for security configuration and whether there are behaviors that we are missing by assuming a similarity between professionals (both programmers, and security engineers) and end users. These groups may achieve their desired security behaviors, and confidence in such behaviors, in very different ways. We suggest this is an important area to investigate further.

5.3 Long-Term Usability Shaping

Continuing the progression from the tactical to the strategic challenges in supporting meaningful choices, we must also consider the long term implications of our usability decisions.

Most discussions of usability typically address the immediate properties of a system, e.g. how long will it take this user to perform a given task? However this is inadequate for our purposes. It is increasingly common to see security usability designs that look fine in the short term, but suffer problems in long term use. Examples include the ‘muscle memory’ of automatically clicking ok to security dialog boxes [23] and alternative password schemes that suffer from interference when they are used by more than one different site. How such systems behave in the laboratory may be a poor predictor for how they behave after months of use and widespread deployment.

In most systems, usability improves with time as users become familiar with the system. Security systems seem to be unusual in that, in many cases, the reverse appears to be true, both in mechanisms like the above and in programming-like configuration systems. As suggested in Section 5.2, such systems are often designed with the assumption that the ‘correct behavior’ is for the user to have thought hard designing their desired configuration and then implemented it. Over time, many small incremental changes erode the original design, in much the same way that ‘coherence’ declines in other software. We need better ways of discussing long-term usability of configuration systems in this regard and how to shape them to support the long term desired, secure, behavior.

There is a related, but more subtle usability effect that occurs over long term use. We are proposing to build systems that allow users to craft their own solutions to problems but as we commented on in Section 3.1 abstractions have a shaping effect, making some things easier than others. We currently do not have a good way of predicting or discussing this effect, but it is important for a number of reasons.

Firstly, and most directly, we want to design systems that encourage generativity – end users appropriating the technology for their own use – in order to do this we need to understand which properties of systems encourage this behavior and which discourage it. As we suggested before, in direct manipulation systems where user interface elements compete for screen space it is hard not to make divergent behaviors more costly in favor of behaviors supported by the abstraction. But this is much less true of programming-like systems which tend to escape the limits of screen space. So there is some hope that we can escape the worst of the normative effect of abstractions by supporting programming, but much further work is needed to understand how to build abstractions that support this. E.g. which features of an API encourage appropriation? How high-level should it be? How can we build abstractions that can be partially deconstructed for reappropriation? How can we structure the API, so it’s easy to find the underlying components? Will consumers mix levels elements from different levels of abstraction? How can we build languages that support this?

Secondly, notations for programming languages tend to live for a very long time and undergo many design increments. Our understanding of how to design programming languages that remain usable after design changes have occurred is limited. There is some very tentative work in this area discussing, for example, the way that *secondary notations* – notations that are not part of the primary syntax, e.g. comments – tend to be appropriated for social uses, which can then be supported by carefully designed computational mechanisms. Indentation in code is an example where this has happened [7, 13]. Another example is the suggested process for evolving notations from the metadesign community who propose a strategy for Seeding, Evolutionary Growth and Reseeding – where an initial system is provided, the behavior of users within the system are observed and then direct computational support is provided for the users operations the users commonly perform [10]. However, while such work on long term usability impact of notations is ongoing, there is still much to do.

Thirdly, and even more subtly, we need to consider the social impact of the primitives we design. In [5] Bowker and Star describe in extensive detail the way that abstraction choice within systems carries political and social implications. When our abstractions are solely in the technical domain, there seems to be little risk. However as we suggested in [4] abstractions for security often carry strong social implications. For example, is the structuring of capabilities and role based access control universally appropriate? The historical studies in both [5] and even Focualt's [11] show that if a technology deploying such an infrastructure became very widely accepted, it would tend to disadvantage those who didn't fit within the system, i.e. groups who didn't think about access management in this way. This is both an important reason to allow end users to construct their own abstractions, as they have the most knowledge of local sensitivities [8], but also a reason to carefully investigate the shaping effects that our primitives have. At the moment, we suggest, we simply don't know whether this is a serious problem or not.

5.4 Spectrum discussions

At a still more strategic level, what we are suggesting is a shift to support a property, generativity, of the technology eco-system. We believe that this is needed for providing users with a meaningful choice in terms of security. However, as we have repeatedly asserted, this is not a black and white matter – there is a spectrum of behavior from generative technology to the appliance, and the ideal place for any given technology to be at along that spectrum is influenced by many factors.

Discussions of spectrums are difficult, and security, usability and socio-technical implications of design are all particularly fraught. In security, for example, some socio-technical norms have been established as to what is an insecure system; say executing arbitrary code from an unknown source with high privilege – there are few cases indeed where we currently wish to do this. But the analogous rhetorical style of categorically declaring things in HCI-SEC as unusable is generally flawed – just because you display a dialog box asking the user a question doesn't automatically mean that it will suffer from the 'muscle memory' problem described previously, and just because the user has to learn something in order to use the system correctly doesn't mean that the designer is engaging in harmful 'train and blame' behavior. Of course, many systems in security are also like this, they would be insecure if deployed outside the context in which they were designed for us. Issues of generativity are also, we suggest, likely to be like this. So the discussions along each of these axes are hard, and simplistic reductions are of little use.

But our challenge is even greater than this, we don't only need to be able to discuss systems along these axes, we need ways of discussing interactions between them. How much generativity are we willing to give up for some extra usability? What are the implications for usability, and generativity, of a security design change which restricts the way in which a primitive might be used?

In order to progress our designs for security and privacy systems in a way that grows a healthy technological eco-system [21], we need to find ways to have nuanced conversations like this.

5.5 Levels of abstraction

A final challenge: at what level of abstraction should we have these design discussions? We have suggested at various points throughout this paper, that various different levels are appropriate. For example, we suggested that designing solutions for specific, concrete, users is a good idea in terms of grounding the design in what a real user may want, but runs the risk of 'over-fitting', and driving us towards building appliances to solve specific problems, with all the problems that brings.

We have also commented that over-abstracting away from the users runs a risk of allowing incorrect assumptions to be introduced into the design, harming usability and security and generatively.

These challenges seem somewhat inevitable, the challenge then is not so much to pick a level of abstraction, but to be prepared to talk about the design of a system at different levels – and to be aware of the risks of doing so. Currently, we have a better understanding of how to build secure, usable systems whilst working with the highly concrete, if we are to support generativity, and meaningful control we need to step back a little and be prepared to have more abstract conversations, but to accept that doing so carries risks.

6. CONCLUSION: THE LAST MILE OF DESIGN

In this paper we have outlined some of the current state of security and usability design practice, and pressures that are driving industry towards building appliances. We have expressed some concerns as to why this is problematic, both in terms of whether it is possible to build systems that offer meaningful choice in this way and whether the type of usability that we achieve by appliancisation is what we want to support our technological eco-system.

We proposed structuring an alternative approach by directly supporting programming activities and have highlighted that there are a number of significant challenges along the way to achieving this, requiring new ways of thinking, talking and designing the usability and security of systems – but at that it offers potential rewards in increased usability, security, generativity as well as the potential for better user experiences.

In most deployed systems, this is too large a challenge to address directly, so we suggest a practical route forwards – to begin easing meaningful choice back into systems by allowing the users to perform the 'last mile of design'. By this, we mean, for any given project rather than providing security configuration systems by either a constrained set of expert-determined opaque presets, or an unusable direct manipulation interface to a programming language, such as huge numbers of tick boxes – think about the problem as if it were a minimal programming language. What properties does it need to support? What would be a helpful way of allowing small, user defined abstractions? What would be a way of allowing the user to define rules over those abstractions, without having to learn a complex syntax? How can you give the user a feeling of confidence in what they have created? How might they go about debugging such a configuration?

We are working on projects doing exactly this, which we suggest may form the beginnings of a new paradigm for security usability.

7. ACKNOWLEDGMENTS

We thank Alan Blackwell and Thomas Green for many invaluable conversations. Luke Church's research is supported by the Eastman Kodak company.

8. REFERENCES

[1] Acquisti, A. and Grossklags J. 2007. What Can Behavioural Economics Teach Us About Privacy?. In Acquisti A., Vimercati S.C., Gritzalis S., Lambrinoudakis C. (eds), Digital Privacy: Theory, Technologies and Practices, Auerbach Publications (Taylor and Francis Group), 363-377, 2007.

[2] Beckwith, L. 2007. Gender HCI Issues in End-User Programming. Doctoral Thesis. Available at http://hdl.handle.net/1957/4954

[3] Blackwell, A. and Burnett, M. 2002. Applying Attention Investment to End-User Programming. In Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (Hcc'02) (September 03 - 06, 2002). HCC. IEEE Computer Society, Washington, DC, 28.

[4] Blackwell, A.F., Church, L. and Green, T.R.G. 2008. The abstract is 'an enemy': Alternative perspectives to computational thinking. In Proceedings PPIG'08, 20th annual workshop of the Psychology of Programming Interest Group, 34-43.

[5] Bowker, G. C. and Star, S. L. 2000 Sorting Things Out: Classification and its Consequences. MIT Press.

[6] Church, L. 2006. Refactored Cognitive Dimensions and Secure Development. Presented at WIP-PPIG'06. Work in Progress Psychology of Programming Interest Group. Available at: http://www.lukechurch.net/Professional/Publications/PPIG-2006-01-RCDsAndSD-Paper.pdf

[7] Church, L. 2007. Tradeoffs in Future Proofing Notations. Presented at WIP-PPIG'07. Work in Progress Psychology of Programming Interest Group. Available at: http://www.lukechurch.net/Professional/Publications/PPIG-2007-01-TradeoffsinFutureProofingNotations-Paper.pdf

[8] Church, L. 2008. End User Security: The democratisation of security usability. Presented at the first international workshop on Security and Human Behavior, SHB'08. Draft available at: http://www.lukechurch.net/Professional/Publications/SHB-2008.pdf

[9] Clarke, S. 2004. Measuring API Usability, Dr. Dr. Dobb's Journal Special Windows/.NET Supplement. Available at: http://www.ddj.com/windows/184405654

[10] Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., and Mehandjiev, N. 2004. Meta-design: a manifesto for end-user development. Commun. ACM 47, 9 (Sep. 2004), 33-37. DOI= http://doi.acm.org/10.1145/1015864.1015884

[11] Foucault, M. 1979. The History of Sexuality: Volume 1. Allen Lane.

[12] Gould, J. D. and Lewis, C. 1985. Designing for usability: key principles and what designers think. Commun. ACM 28, 3 (Mar. 1985), 300-311. DOI= http://doi.acm.org/10.1145/3166.3170

[13] Green, T.R.G. 2006. The Hindsight Saga, Keynote address presented at VL/HCC, PPIG'06, 18th annual workshop of the Psychology of Programming Interest Group

[14] Green, T.R.G and Petre, M. 1996. Usability analysis of visual programming environments: a "cognitive dimensions" framework. Journal of Visual Languages and Computing, 7:131-174.

[15] Hoofnagle, C. J. and King, J. 2008 ,Research Report: What Californians Understand About Privacy Offline (May 15, 2008). Available at SSRN: http://ssrn.com/abstract=1133075

[16] Jenson, S. 2002 The Simplicity Shift: Innovative Design Tactics in a Corporate World. Cambridge University Press.

[17] Jobs, S. 2007. Macworld San Francisco 2007 Keynote Address, Jan. 9, 2007, available at http://www.apple.com/quicktime/qtv/mwsf07/

[18] Ko, A. J. and Myers, B. A. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In Proceedings of the 30th international Conference on Software Engineering (Leipzig, Germany, May 10 - 18, 2008). ICSE '08. ACM, New York, NY, 301-310. DOI= http://doi.acm.org/10.1145/1368088.1368130

[19] Ko, A. J. Abraham, R. Beckwith, L. Blackwell, A. Burnett, M, Erwig, M. Lawrence, J. Lieberman, H. Myers, B. Beth Rosson, M. Rothermel, G. Scaffidi, C. Shaw, M. and Wiedenbeck S. (in press). The State of the Art in End-User Software Engineering. Accepted for publication in ACM Computing Surveys.

[20] Maeda, J. 2006 The Laws of Simplicity (Simplicity: Design, Technology, Business, Life). The MIT Press.

[21] Nardi, B. A. and O'Day, V. L. 1999 Information Ecologies: Using Technology with Heart. MIT Press.

[22] Preece, J., Rogers, Y., and Sharp, H. 2002 Interaction Design. 1st. John Wiley & Sons, Inc.

[23] Schneier, B. 2006. Microsoft Vista's Endless Security Warnings. Available at http://www.schneier.com/blog/archives/2006/04/microsoft_vista.html

[24] Shapiro, C. and Varian, H. R. 1998 Information Rules: a Strategic Guide to the Network Economy. Harvard Business School Press.

[25] Shneiderman, B. 1987. Direct manipulation: A step beyond programming languages. In Human-Computer interaction: A Multidisciplinary Approach, R. M. Baecker, Ed. Morgan Kaufmann Publishers, San Francisco, CA, 461-467.

[26] Subrahmaniyan, N., Beckwith, L., Grigoreanu, V., Burnett, M., Wiedenbeck, S., Narayanan, V., Bucht, K., Drummond, R., and Fern, X. 2008. Testing vs. code inspection vs. what else?: male and female end users' debugging strategies. In Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems (Florence, Italy, April 05 - 10, 2008). CHI '08. ACM, New York, NY, 617-626. DOI= http://doi.acm.org/10.1145/1357054.1357153

[27] Whitten, A. and Tygar, J. D. 1999. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. In Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8 (Washington, D.C., August 23 - 26, 1999). USENIX Security Symposium. USENIX Association, Berkeley, CA, 14-14.

[28] Zittrain, J. 2008 The Future of the Internet--And how to Stop it. Yale University Press