

Position Paper on the “Science of Design”

Greg Morrisett
Cornell University

October 1, 2003

1 My Background

My research interests are in programming language design, semantics, and implementation. Most of my career has focused on the application of type systems and type theory to practical programming issues. I worked on type systems for compiler intermediate languages to better support optimization. I also developed the ideas behind typed assembly language (TAL) and pushed the agenda of a type-preserving, and more generally proof-preserving compiler (also known as a certifying compiler.)

Work in this area inspired (and happened concurrently with) the development of the more general framework of proof-carrying code (PCC). Today, my interests are in automatically achieving (machine-checkable) proofs of rich properties for real code, going well beyond the simple type safety guarantees of TAL. In particular, I have focused on various security policies and attendant implementation techniques such as inlined reference monitors.

2 My Take on the “Science of Design”

Although “Science of Design” (for software-intensive systems) is intended to be broad and all-encompassing, I have found it difficult to identify concrete research goals that would actually address the broad set of issues, ranging from engineering to economics to ergonomics to society. So I will focus on some notion of “dependable” or at least “reliable” software, taking into account engineering and economic constraints.

Perhaps the first thing we should do is take stock of the design principles, techniques, and patterns that researchers and practitioners claim lead to dependable software. Then we should identify why they *don't* work, in hopes of uncovering new principles, techniques, and patterns that will work. Here, I will discuss a couple of key principles.

3 KISS

Let's start with the grandmother of all engineering principles: “keep it simple, stupid” (KISS). Sadly, software is rarely simple. Windows XP is over 50 million lines of code. Is it any wonder

that we don't consider it reliable? The reason Windows (and similar systems) grow to such size is due to simple economics: Adding new features causes you to upgrade, and there's very little incentive to remove old features or to bulletproof the code. The truth of the matter is that Microsoft and most vendors follow a higher principle: "keep it worth buying".

The trend is changing, as evidenced by Microsoft's Trustworthy Computing Initiative. I believe that economic and legal pressure will start forcing software vendors to place a premium on reliability (or at least the appearance thereof). This means that a central design question will be how to re-factor and re-engineer 50 million lines to make it dependable, but still "keep it worth buying".

This will require new architectures at both the hardware and software levels (see for instance NGSCB a.k.a Palladium) to isolate components and secure communications. Note that we will need to do this kind of refactoring not only for desktops and servers, but also for the Internet and its underlying protocols. This will demand radical ideas (e.g., "Throw out TCP in favor of an authenticated, reliable communication service", "Throw out legacy device drivers") that today seem heretical. These big changes won't happen tomorrow, but we need to be thinking about (a) what kind of architectures we would like to have and (b) how to move the existing systems towards those goals.

4 Least Privilege

In the security literature, the principle of "least privilege" is considered central: Principals should be given only those rights and capabilities needed to accomplish their appointed task. Least privilege sounds good on paper, but it's almost never realized because it demands too much work on the part of users. Do you manage to set the permissions on each and every file? Do you re-configure your Web browser's security settings before clicking on a link? Do you run each process using a different account, configured so that the process can only read/write/execute a small set of files? We almost never do this because it gets in the way of useful work. In other words, least privilege is really a denial of service attack.

Actually, there's nothing wrong with the principle of least privilege, but our current architectures don't support it very well. In part, this is because we need some way of taking the user out of the decision-making process most of the time.

We need new architectures that can analyze code to determine what its behavior might be when run, and then we need auditing mechanisms that allow us to track enough context to determine whether or not we deem a behavior to be acceptable. Technical ideas like proof-carrying code make it possible for us to, at least in principle, construct "semantic" signatures for code. Other techniques, like code rewriting and interpretation, allow us to interpose context-specific policies. If, in addition, we had history information about data and programs (e.g., this Word document was pulled off the Internet from an untrusted domain) then we can bring these tools to bear (e.g., turn off macros) without involving humans in the decision making process. In turn, this will allow us to better realize a least privilege environment.