

# High Assurance Software Development

David Burke, Joe Hurd and Aaron Tomb  
Galois, Inc.

`{davidb,joe,atomb}@galois.com`

August 12, 2010

## Abstract

The purpose of this paper is describe how to make software assurance a part of a science of security. Software assurance as practiced is a grab-bag of techniques, heuristics, and lessons learned from earlier failures. Given the importance of software to critical infrastructures (electricity, banking, medicine), this is an untenable situation; the smooth functioning of our society depends on this software, and we need a more rigorous foundation for assessments about the trustworthiness of these systems.

In this paper we present an evidence-based approach to high assurance, in which diverse development teams can communicate in a common language to tackle the challenges of developing secure systems. Furthermore, this framework supports formal inference techniques (in particular, a *trust relationship analysis*), so that we can use automated reasoning to deal with scalability issues. Perhaps most importantly, an evidence-based approach lets us tailor the tools that we bring to bear on each claim: formal methods: testing; configuration management; and so forth all have their place in an assurance argument. In the end, it's all about deploying systems where the residual risk has been minimized, given finite resources and time. Understanding this and managing it effectively is what the science of security is all about.

# 1 Introduction

## 1.1 Problem Statement

The purpose of this paper is describe how to make software assurance a part of a *science of security*. Software assurance as practiced is a grab-bag of techniques, heuristics, and lessons learned from earlier failures. Given the importance of software to critical infrastructures (electricity, banking, medicine), this is an untenable situation; the smooth functioning of our society depends on this software, and we need a more rigorous foundation for assessments about the trustworthiness of these systems.

“Given the issues with today’s software engineering, how can we build the systems of the future that are likely to have billions of lines of code?”

The quote above is from a 2006 DoD-commissioned report from SEI (Software Engineering Institute) entitled “Ultra-Large-Scale Systems: The Software Challenge of the Future” [13]. Ultra-Large-Scale (ULS) Systems are essentially *systems of systems*, comprised of a tremendous number of components, with complex interconnections between them. These components are typically built by different teams at different times, making integration a serious challenge. In addition, ULS systems have a wide variety of stakeholders, making it difficult to agree what the system’s goals should be. Lastly, they are *socio-technical* systems: people aren’t simply users set apart from it—they are an integral part of it. All of these factors make ULS systems a challenge that is quite different from the traditional picture of software development, with a team dedicated to the design and implementation of a well-defined piece of software.

The challenges presented by ULS systems are relevant to any discussion of software assurance, because of the inexorable trends towards greater size and complexity of software. A cliched, but useful example is the growth of the Windows OS. When Windows 95 was released, it was considered a behemoth at 15 million lines of code, but it is dwarfed by the estimated 50 million lines of code in Vista.

Growth in terms of lines of code isn’t the only change in the way software has been designed; a few decades ago, systems were more likely to be standalone, instead of networked, so their environments were more constrained,

and predictable. Compare this to the current situation—most mission critical software doesn't operate in a static environment, with well-defined inputs. Instead, we see a steady increase in the number and complexity of 'software agents', which are programmed with some rudimentary ability to make judgments, and continuously interacting with large numbers of other human and software agents.

A somewhat flippant, but accurate way of characterizing these changes is to say that "today's ULS system is tomorrow's web application". And clearly, this growth in size and complexity makes life exponentially more difficult for both designers and evaluators to be able to give assurance guarantees. But the reality is that we still have to make judgment calls whether a system is trustworthy enough to deploy. So how is this done now, and how well is it working?

## 1.2 Current Evaluation Practice

The current international standard for security evaluation is the Common Criteria (ANSI 15408), an outgrowth of the influential 'Orange Book' of the 1980's.<sup>1</sup> It provides various *Evaluation Assurance Levels* (EALs), from EAL1 ("functionally tested": some testing for correct operation occurs, but there are no serious security threats to consider) to EAL7 ("formally verified design and tested": the system is designed with well-defined security functionality that is amenable to verification through formal methods).

The evaluation process, in a nutshell, is something like this: a vendor specifies the security requirements that the system is responsible for upholding, and the level of assurance that they wish to claim for the system. Then, once the design and implementation is complete, the vendor delivers a sample unit of the system to the evaluation team, along with a big stack of paper containing the documentation of everything from test reports to developer bios to a printout of the source code. The job of the evaluator is to figure out

---

<sup>1</sup>The DoD and IC (Intelligence Community) have their own internal process for security evaluation, which is called Certification and Accreditation (C&A). Certification is not dissimilar to Common Criteria evaluation in the sense that the goal is to take a look at how well the system enforces a security policy against a declared set of threats. The differences lie in the output, and the use to which it's put: in CC, you either get a 'yes' or 'no', whereas in certification, the output is a writeup of the findings of their evaluation of the system. This evaluation is passed on to an accreditation official, who gets to make the call on whether the system can be deployed or not in a situation, given the vulnerabilities the certification team found.

how to use the sample unit plus all the development documentation (and the CC guidelines) to properly characterize its fitness in real-world conditions.

The concept of ‘trust’ comes up very naturally in discussions of security and assurance, both in the words ‘trusted’ and ‘trustworthy’. These words have technical definitions that differ from their vernacular usage. In a security context, when you deploy a system that is responsible for enforcing an aspect of a security policy, it is trusted. Another common way of saying this is that any component/system that has the potential to break your security policy is by definition ‘trusted’. In contrast, ‘trustworthy’ refers to whether that trust is warranted. For example, you may deploy a security-enforcing component that hasn’t yet been tested comprehensively. That component is trusted, but whether or not it is worthy of that trust (trustworthy) is another matter entirely.

Another useful distinction between these two words that have the same root is that one refers to an action, whereas the other refers to a belief. The process of producing evidence is establishing the confidence in your belief of the trustworthiness of a system, in preparation for making the decision whether to trust it.

### **1.3 Difficulties with Current Practice**

This section looks at three reasons why the current practice isn’t effective.

#### **1.3.1 Standards Aren’t Applied in a Consistent Manner**

In the CC paradigm, the evaluators are commercial 3rd party entities. On paper, this sounds like a good idea: use free market principles to make the process more efficient. In practice, it hasn’t turned out this way, and there is evidence that this has led to vendors shopping around for the most lenient evaluation party [4].

There is the similar concern that evaluators are confronted by systems with literally millions of lines of code in them. There isn’t the time to exhaustively canvass every line, so different evaluation teams are likely going to make different decisions as to how to spend their evaluation resources. But this lowers our confidence in the conclusions reached.

### 1.3.2 CC Evaluation Works Against Commercial Norms

Success in the marketplace often seems to mean “release the software now to get ‘first-mover’ advantage, and fix bugs later”. Many companies issue a steady stream of bug fixes and capability updates, in order to respond to issues found in the field, and keep customers locked in. But in the CC paradigm, the product isn’t brought into evaluation until it is completely finished. Developing products this way is likely to be commercial suicide.

### 1.3.3 The Process Isn’t Scalable

One phrase we’ve heard that captures the essence of the current paradigm is that “evaluation is a social process”, which is another way of saying that it is time-consuming and expensive. The government agencies who oversee the evaluation process are straining to keep up with the demands on their time. As software systems get larger and more complex, more resources will be needed to be applied to each evaluation, and the whole process just won’t scale.

## 1.4 “What About X as a Silver Bullet?”

Two approaches are often put forward as a solution to the assurance problem.

### 1.4.1 Process

Ever since the beginnings of software engineering as a discipline, there have been evangelists for the assertion that “if you get the processes right, you’ll get the right outcomes”. There seems to be a lot of truth to this statement if you’re discussing, say, the manufacturing of reliable automobiles. And so by analogy, there is a hope that in the not-too-distant future we’ll have *software factories* turning out robust components that can be integrated with other similarly trustworthy pieces to build the dependable systems of the future. But the evidence that this vision will come to pass is underwhelming. The skepticism can be summed up in the following two quotes:

“I run your code, not your process”

“Dirty water can flow through a clean pipe” [3]

A more modest, but realistic role for process would be to say that good processes are necessary, but not sufficient. It’s very hard to imagine that a

trustworthy system of any complexity could be developed by a team with, say, no configuration management process. But on the other hand, just having processes in place to support your development infrastructure isn't going to make the resultant system trustworthy, by itself.<sup>2</sup>

#### 1.4.2 Formal Methods

We have a very attractive vision of being able to carry out computer-based proofs of the correctness of software. There was a great deal of interest and optimism in the 1970's in the belief that formal methods would solve the problem of program correctness, and give us dependability guarantees for systems. This hope turned out to be premature, and even though formal methods have made huge strides in the interim, the goal of verifying the correctness of a complex software-based system (say, a million lines of code) is still out of reach.

As these early, optimistic claims didn't pan out, there was a backlash against formal methods, and proponents today are more careful not to inadvertently oversell formal methods as a panacea.

## 2 Galois High Assurance Methodology

The state of software assurance can be summed up as: “we're muddling through”. But we believe that the current paradigm is running up against limits. It's too expensive in terms of time and resources, and there is very little reason to believe that the results are scientific, in the sense of being based on some rigorous foundations, and allowing for consistency and scalability.

We claim that dependable (or ‘high assurance’) software can become the norm, if a science of security is brought to bear on the software assurance problem. In this section, we describe what the elements of that science look like.

### 2.1 Defining High Assurance

In order to discuss the concept of high assurance, it's important to have a working definition of what you mean in the first place. How do you ‘do’

---

<sup>2</sup>Unless your process is the magic meta-process: “do what is needed to guarantee completely trustworthy software”

high assurance development? We’ve heard various stabs at this, such as “putting in the extra 10%”, “documenting everything you do” or “build it to withstand malicious attacks”, but none of these capture the essential idea. Here’s our definition:

**High assurance development means producing compelling evidence that a system meets specified requirements.**

The first thing to recognize from this definition is one of the key misconceptions when discussing high assurance: thinking that high assurance is a property of the system itself. As the definition says, high assurance is all about the *evidence* that the system meets specified requirements. You can’t say that something is high assurance just because you believe that you took great pains to do a robust design and faithful implementation of that design. It’s the evidence that makes it so.

The phrase “compelling evidence” implies a specific entity who is doing the evaluation, just as the phrase “specified requirements” makes clear that the evidence is against a set of concrete claims. That is, the notion of high assurance is context-dependent. It’s common to hear things like “we’re going to build a high-assurance system”, but this is a meaningless statement without a context. What are the specific threats against this system? In what environment will it be operated in? What specific security claims are you making for it?

## 2.2 Thinking about Requirements

Security engineers generally derive the system requirements from the anticipated threats to the system. It is a waste of time and resources to build security-enforcing capabilities into a system for an irrelevant threat.<sup>3</sup> But it’s also important to realize that there are two fundamentally different kinds of security engineering requirements: *design requirements* and *implementation requirements*.

As an example of this difference, imagine that you are concerned about an eavesdropping threat over fiber optic cables. A possible decision would be to encrypt the traffic using AES (say, after deciding that physically securing the cables isn’t feasible). This encryption decision is one of design, and how

---

<sup>3</sup>One of the stimulating challenges inherent to security engineering is deciding whether a threat is relevant or not!

you implement this decision is of equal importance—perhaps your AES implementation is flawed, or you’re using a faulty random number generator. Therefore, we insist on evidence that addresses both design and implementation when making high assurance arguments.

However, instead of talking about requirements, and the evidence that those requirements are being met, we prefer instead to translate requirements into ‘claims’ for the system. The main reason for this is that in many cases, requirements are written in a bloodless, impersonal manner, often in the form of commandments: “the system should/shall/will”. Claims are more naturally written in the form of a contract, which lends itself to more concrete, specific, and direct language that captures the idea that the system is taking action to mitigate a threat. So for example, instead of saying that we have an encryption requirement for message traffic, we make a claim like “message traffic is protected against eavesdropping”, or even better “message traffic is protected against eavesdropping through AES encryption”.

The heart of our approach, then, is the pairing of concrete, relevant claims and compelling evidence, where those claims have been derived from the threats confronting the system. We use the claims, and the evidence for those claims, to assess the residual risk to the system, given the threats confronting the system.

## 2.3 Residual Risk

Our security engineering goal is to minimize the residual risk. In order to do this, we need to mitigate the system threats, and to do so, we make certain claims about the system’s ability to do so. The believability of these claims is dependent on the strength of the evidence. So we can imagine that the *normalized claim* is equal to the strength that a claim addresses a threat multiplied by the strength of the evidence that supports the claim. The difference between the aggregate value of the claims and the aggregate value of the threats is the value of the residual risk.

We can represent this relationship as follows:

$$\text{residual risk} = \text{threats} - \text{claims} * \text{evidence}$$

One objection to a quantitative approach like this one is to claim that there just isn’t enough data to provide the estimates needed to make the technique work. For example, what is the probability of the threat of insider

attacks at a military installation. Is it 1 in 1,000? 1 in 10,000? How does the number vary with respect to the clearance granted? Our response is that we don't need exact numbers to make this approach useful, for two reasons:

1. In many cases, empirical data does exist that can give us a ballpark, or rough-order-of-magnitude estimate. And the less sure we are about the precision of the data, the more important it is for us to run a sensitivity analysis to see how much the residual risk varies in relation to changes in our estimates. Producing our residual risk numbers as a range, instead of a point estimate, is a good modeling practice.
2. Just as important is the fact that much of the benefit of this approach isn't the actual number we get at the end, but making explicit the assumptions that go into the analysis.<sup>4</sup> If, for instance, an evaluator disagrees with your estimate about the prevalence of an insider threat, it is an explicit parameter than can be altered, and the consequences understood.

And in many cases, we do have empirical data that can help us. Our thinking about residual risk is inspired by three *Axioms of Insecurity* that appeared in the National Academy of Science book "Trust in Cyberspace" [15]. They are:

1. Insecurity exists.
2. Insecurity cannot be eliminated.
3. Insecurity can be moved around.

We have rewritten these to capture our evidence-based approach to minimizing residual risk in a system:

1. Residual risk exists.
2. Residual risk cannot be completely eliminated.
3. Residual risk can be reduced through effective use of evidence.

There are two important implications of this approach:

---

<sup>4</sup>cf. Hamming's famous quote: "the purpose of computing is not numbers, but insight".

1. You need a method that links the threats and the associated claims: a way to translate the statements of threats into security enforcement obligations (the claims) throughout the system. We call this process a *Trust Relationship Analysis*.
2. You need a method that unifies the diverse kinds of evidence generated in support of claims, so that a quantitative measure of the strength of claims and strength of evidence can be calculated. This process of structuring evidence against claims is called building *Security Assurance Cases*.

The combination of a Trust Relationship Analysis and a set of Security Assurance Cases allows us to determine the residual risk in a system in a rigorous, quantitative manner. In the next two sections, we discuss each of these methods.

## 2.4 Trust Relationship Analysis

Before we make an assurance case for a security system, we need to understand the threats that the system is designed to prevent. We do this by analyzing the system in terms of what we call *trust relationships*: the security-based properties that the system is going to enforce. In the next section, we introduce the concept of trust relationships through a simple example, and then follow up with a discussion of how this approach would scale up, in practice, to a realistic security system.

### 2.4.1 A Safe Example

Suppose that we have a flash drive that contains important data, and we wish to prevent that data against the threat of unauthorized modification. (i.e., protect the integrity of the data). That is, we want to build a system whose overall security policy can be described as “protect the flash drive against unauthorized modification”. After making an assessment about the seriousness of the threat, we decide to mitigate it by physically protecting the flash drive. So we purchase a strong safe from a reputable manufacturer, and give the combination to the site security officer (SSO), who is responsible for not sharing that combination with anybody.

At this point in this discussion we have identified two properties:

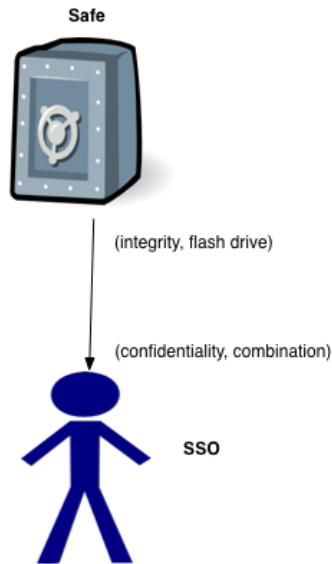


Figure 1: A trust relationship between two security properties.

1. The safe has the obligation to enforce the integrity of the flash drive.
2. The SSO has the obligation to enforce the confidentiality of the combination.

Figure 1 illustrates the *trust relationship* between these two properties. We can say that, with respect to the enforcement of property 1 (an integrity property), we are trusting that property 2 (a confidentiality property) will be upheld. This is what we mean when we say that there is a *trust relationship* between the two properties. If property 2 isn't trustworthy (say, the SSO shares the combination with his drinking buddies, or writes it down in plain view of colleagues), then we have much less confidence that property 1 will be enforced.

Suppose that further evaluation of the threats to our system reveals that we need to take seriously two additional threats:

1. Somebody might attempt to gain unauthorized access to the safe, by either figuring out the combination (think stethoscope!), or by blowing the safe door open with explosives (maybe destroying the flash drive in the process, which certainly would be an integrity attack!).

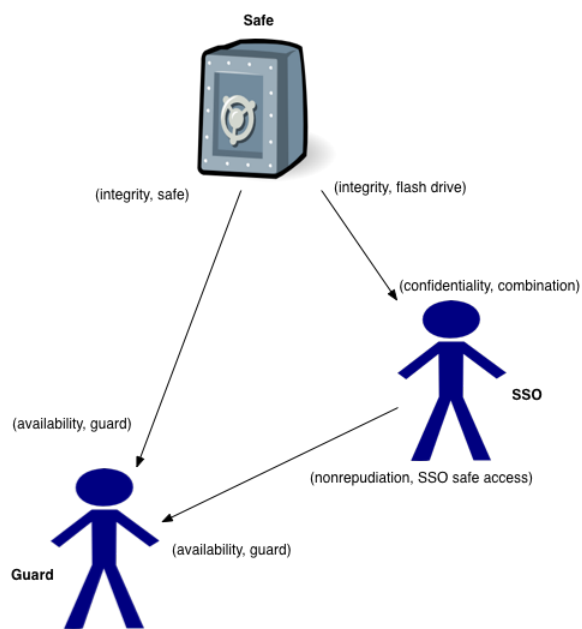


Figure 2: Trust relationships in the extended system.

2. The SSO engages in an insider attack, using the combination to sneak into the safe, and make changes to the data.

To mitigate these threats, the system is extended to place a guard in front of the safe. The guard doesn't have access to the safe combination, but the presence of the guard will deter threat 1. In addition, the guard can keep records of whether the SSO ever accessed the safe. This gives us the following additional properties to reason about:

- The availability of the guard.
- The integrity of the safe.
- Non-repudiation of the SSO's safe accesses.

Figure 2 illustrates two extra trust relationships in the extended system: one from the safe to the guard (the integrity of the safe is enforced by the availability of the security guard); and one from the SSO to the guard (nonrepudiation of safe access is enforced by the availability of the guard). Further refinements of the system can be made (do we need to worry about the SSO suborning the guard?) until we are satisfied that we have captured all of the relevant trust relationships in the system.

#### 2.4.2 Trust Relationships in General

In the safe example of the previous section, all the properties could be written as *property-of-asset*, where *property* is one of the canonical security properties:<sup>5</sup>

- Confidentiality
- Integrity
- Availability
- Authentication
- Non-repudiation

---

<sup>5</sup>Sometimes called the *CIA* properties after the acronym formed by the first three.

For a general system, where do these properties come from? Many textbooks on security engineering will suggest that doing a threat assessment is one of the first steps in any security engineering initiative [5, 12]. We find that the identification of properties follows naturally from the threats, as in our safe example. Given a system and its security policy (i.e., a high-level statement of the overall security goals of the system), we perform a trust relationship analysis by decomposing the overall system security policy into a set of trust relationships which enforce those obligations. As trust relationships are defined, this tends to generate more threats (for instance, in our example, now that we've identified a guard as being a system agent, we might want to consider threats against the guard being incapacitated, or distracted, etc.). So a trust relationship analysis ends up being a method for doing a threat assessment.

Trust relationship analysis lends itself to an iterative approach, where the decomposition suggests a preliminary security architecture, and as the trust relationships are defined, we can make assessments as to whether those obligations seem realistic (“you want us to hire 987 security guards?”). Just as importantly, this technique allows us to take into consideration the severity of the threats against the system. Suppose each threat is assigned a number from 1 to 10 (where 10 represents the greatest risk, by some appropriate metrics involving severity and probability) Since each trust relationship can be associated with the threat that it is mitigating (and there may be multiple threats), this allows us to associate with each system component some kind of metric as to its criticality for the system’s ability to enforce the security policy.<sup>6</sup>

At Galois, we have found that one of the immediate benefits of doing a trust relationship analysis is that it makes explicit the assumptions and relationships between the threats, and the various components of the system that are meant to mitigate those threats. Gaining a shared understanding of the overall security architecture is a necessary first step in being able to have an effective conversation on what changes to the design might be necessary to obtain the desired level of trustworthiness in the system.

A trust relationship analysis also lends itself to a high degree of formalization, which is likely to be an important consideration when dealing with complex systems. In particular, we are finding that applying various modal

---

<sup>6</sup>The component criticality metric could be maximum, average, sum, or a combination of the associated threat severity metrics.

logics to trust relationships enables us to make inferences about system security that wouldn't be possible from a visual inspection of the system. This is an area of active research at Galois.

## 2.5 Security Assurance Cases

As alluded to earlier, various approaches to security assurance have been championed over the past few decades. For example, you could make formal methods central to your approach (tapping into the power of mathematics), or alternatively, you might champion some kind of process-based approach (where the underlying metaphor is that gaining high assurance is akin to quality control on a manufacturing line).

We advocate an evidence-based approach to assurance. In this framework, the underlying idea is that of making an *assurance case*, much like a lawyer prepares a legal case. For example, in a legal setting, a lawyer makes an argument that is based upon a variety of types of evidence: physical evidence, eyewitness testimony, and so forth. It is up to the lawyer to make clear how each piece of evidence is relevant, credible, and supportive of his/her case.

Likewise, in our framework, we look at each piece of evidence in the context of the overall assurance argument. It's not enough to just say “we executed rigorous processes” or “we have a proof of the correctness of this algorithm”—we need to think about what specific claims each piece of evidence addresses, and then to assess the relevance, credibility, and force of that evidence against that claim.

Various proposed solutions to high assurance development (“test the hell out of it”, “make sure that you have the right processes in place”, “ensure that your people have training in secure SW development”) just become different kinds of evidence. For example, process evidence would provide evidence against other some kinds of claims (for instance, “due to our configuration management processes, it is very unlikely that somebody outside the project team could have modified the source code”). Formal methods can be used as a tool to generate compelling evidence supporting the claim that key algorithms have been implemented correctly. And producing evidence of ‘pedigree’ (“we have engineers on this project who have worked on similar projects that have been successfully deployed without serious incident”) and ‘provenance’ (“our code is built on top of X, which has been in use for 10 years without demonstrating any access control vulnerabilities”) don't sound like special pleading in this framework: they simply become specific types of

evidence for your case.

If we truly want a science of security, then we need an assurance approach that is quantitative, repeatable, and transparent. We believe that the following three key tasks accomplish this:

**Evidence Generation** Formal proofs, test documentation, design documentation, process documentation, and so forth.

**Evidence Organization** What claims are we making for the system, and how does the evidence line up against those claims (for instance, a particular piece of evidence may support numerous specific security claims)? How do we assign a quantitative measure of the residual risk in the system?

**Evidence Presentation** How do we make it clear to the evaluators what the overall strength of our assurance case is? How do they interact with the assurance case, in order to home in on areas where the residual risk is too high or to engage in “what-if?” scenarios?

If we’ve done this properly, we’ll end up with an assurance case that satisfies the *three C’s*: an effective argument is:

**Clear** Evaluators understand the assurance argument, and can adequately assess its merit; it is clear where residual risk resides in that system, and what the severity of that risk is.

**Comprehensive** The assurance argument addresses the whole spectrum of relevant threats to the system.

**Compelling** The characterization of the system’s residual risk (in terms of both location and severity) is believable to the evaluator.

The whole process is designed to be transparent to the evaluator, and as we’ll see, is both quantitative and repeatable. Later in this paper, we’ll discuss *Evidence Generation* in more detail, by focusing on some powerful, analytically-based tools used at Galois. In the next section, we introduce the notion of a *Toulmin structure*, which is the basis for addressing *Evidence Organization* and Presentation.

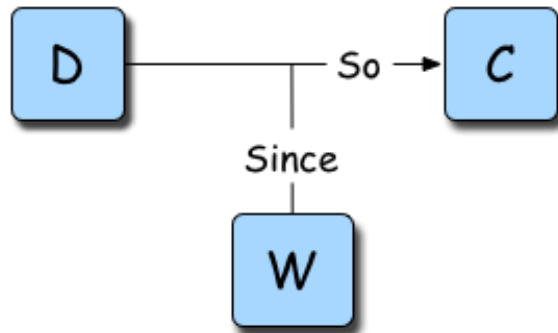


Figure 3: A simple Toulmin structure.

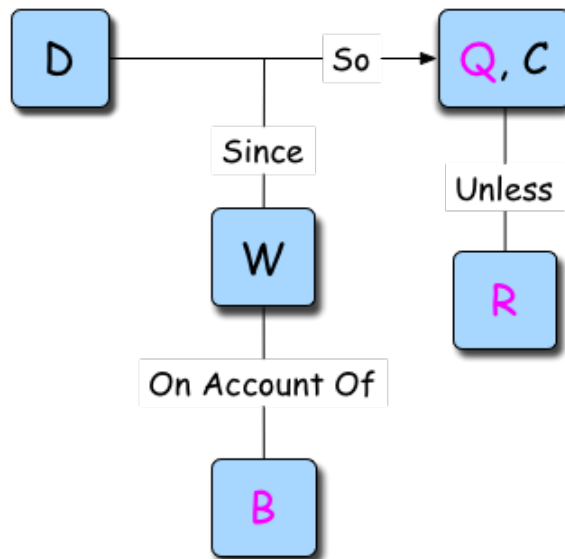


Figure 4: A full Toulmin structure.

### 2.5.1 Toulmin Structures

Our framework is inspired by work done by the philosopher Steven Toulmin, whose work on argumentation in the real world has been extremely influential in the legal fields, and has been gaining traction in the safety community in recent decades [16]. The thrust of his work is to argue that real-world arguments rarely conform neatly to syllogisms (which have historically been held up as the epitome of rational argumentation). Instead, Toulmin proposes what is now known as a *Toulmin structure*, the heart of which is made up of three pieces:

1. A claim.
2. Evidence for that claim.
3. Justification as to why the evidence supports the claim.<sup>7</sup>

Figure 3 illustrates a simple Toulmin structure. (Toulmin recognized that real-world arguments can be more complex than this; an example of a full Toulmin structure is pictured in Figure 4.) It is possible to challenge the justification; a Toulmin structure handles this by treating the justification as an additional claim, which can be supported by its own Toulmin structure. This recursive process bottoms out when the justifications are judged to be axiomatic in the context of the argument (for example, a fact being stipulated).

### 2.5.2 Assurance Cases

We construct our assurance argument by producing a set of Toulmin structures. The idea is to start with a claim, and to decompose the claim into a set of sub-claims. So, in our safe example, the starting claim might be “The safe protects the integrity of the flash drive stored inside”. We then take that claim, and break it down into the following sub-claims:

1. The safe cannot be removed from the site.
2. Any forcible entry attempt will be detected.

---

<sup>7</sup>A justification was originally referred to as a “warrant”. Similarly, in Toulmin’s formulation, evidence was referred to as “datum”.

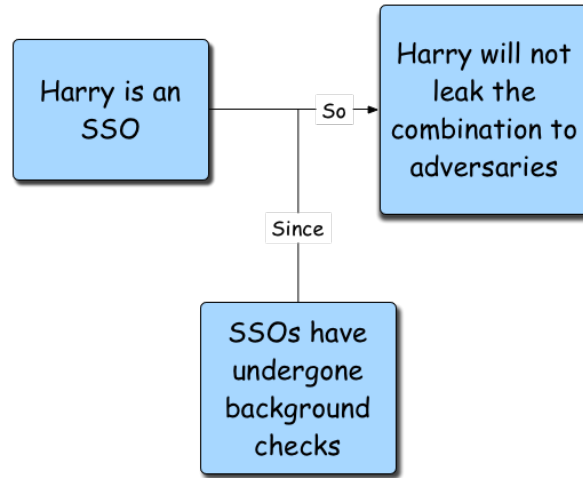


Figure 5: A Toulmin structure for the safe example.

3. The only person who can gain non-forcible access to the safe is the SSO.

We then look at each of the sub-claims, and decide whether they are independent of each other with respect to the higher-level claim. In this example, we see that sub-claims 2 and 3 have an ‘and’ relationship with each other: both of them must be true for the higher-level claim to hold. We denote these on a diagram by the shape of the box containing the sub-claim, as illustrated in Figure 5. Each of these sub-claims is then itself decomposed, until the procedure ceases to be a useful exercise (in the judgment of the modelers). So taking sub-claim 3 as an example, we could make the following sub-sub-claims:

1. The SSO will not willingly share the combination with anybody.
2. The SSO will not make any record of the combination where it can be inadvertently shared.
3. A guard will be watching the safe at all times, so that only the SSO accesses the safe.

Each of these sub-sub-claims, like the ones at a higher level, suggest the kinds of evidence that might justify the claim. So, for the second one, we might

introduce as evidence the following two facts:

1. The SSO has had training about the dangers of writing down a safe combination.
2. The SSO has never had any incidents in which it was discovered that he/she was writing down a combination.

One of the ways that we know that we're done with our recursive process of refining claims is when the evidence to support the claim seems like the right level of granularity. So in the safe example case, knowing that we have access to somebody's security training record would encourage us to say "we can stop here: we think we have evidence that lines up well with the claim". Contrast this situation with one in which a claim is very broad, for example: "this information will not leak outside the organization". In this case, we'd likely need to refine the claim before feeling confidence that our evidence is, or could be adequate to justify the claim. And as mentioned previously, we would expect to produce a set of claims and associated evidence to cover both the design and implementation of the system.

### **2.5.3 Putting it All Together**

Once we've refined the claims appropriately, and assigned evidence to each claim, we need a way of assigning a value to the strength of evidence, and a way to combine the strength of evidence as we move up from sub-claims to claims. There are at least two ways of doing this:

1. Organize the kinds of evidence on a lattice, and assign an ordinal value to each evidence type. So, for example, a simple lattice might contain 4 kinds of evidence, arranged in a diamond shape. At the top is formal methods, which is stronger than both testing and process (which are considered incommeasurable), and both testing and process are considered stronger than, say, evidence from background checks on users or developers.
2. Use probability theory to assign a subjective belief between 0 and 1 (where 1 means that the evidence makes the claim 100% trustworthy, and the opposite for 0).

As far as combining evidence, there are various ways of doing so. For instance, given three independent sub-claims, you might say that the combined evidence is the maximum of the evidence of the sub-claims. Or, perhaps take the minimum if you're being conservative. Or the average - the point here is not so much that there is a right answer, but that you're applying a technique consistently, and there is transparency in the way you combine evidence. In addition to the simple ways we've described, there are some well-known, and more sophisticated methods of combining evidence that can be applied to assurance cases, most notably Dempster-Shafer theory.

## 3 Evidence Generation Technology

### 3.1 Process

It has been observed that to obtain an EAL4 rating for a system, it isn't necessary to do any specialized security engineering: just good solid software engineering will get you there. But the higher EALs, with their requirements for more formal specification and formal proof, are an acknowledgement of the power and potential of formal methods.

That doesn't mean that other kinds of evidence aren't important, too. It is true, although not recognized often enough, that factors like 'pedigree' and 'provenance' can be very persuasive aspects in a portfolio of evidence. Human nature being the way it is, it is natural (and sensible) to give weight to the fact that the team who produced system  $B$  is essentially the same team that produced similar system  $A$  ten years ago, and  $A$  has been deployed in mission-critical situations with no serious failures. Clearly, this team has a strong pedigree, and we believe that this ought to be made explicit in the assurance argument.

Similarly, if a team is using an off-the-shelf piece of code in their system, and it can be shown both that the code hasn't been altered and the code hasn't demonstrated any serious vulnerabilities, then we have a provenance-based piece of evidence for the security claims associated with this code. Again, this claim is made explicit in the assurance argument.

## 3.2 Formal Methods

As useful as these kinds of evidence can be to an assurance case, it still remains the case that formal methods can bring a level of confidence in certain claims that, say, testing cannot match. The popular conception of formal methods is that they are “heavyweight”, and cost too much in terms of time and resources. But this is not the case. At a fundamental level, all formal methods are just models of a system, and successful security engineering is about having the right models.<sup>8</sup> The art of applying formal methods is the art of applying the appropriate models.

For example, imagine that our trust relationship analysis shows that the key residual risk in a system resides in a single component.<sup>9</sup> One strategy might be redesign the system to move the residual risk elsewhere, but after evaluating alternative designs in terms of residual risk reduction, we might decide to apply theorem-proving-based formal methods to this component. Even though theorem proving can be very resource-intensive, the amount of bang you get for the buck (in terms of reducing residual risk) will quite often justify the expense. At Galois, formal methods are the most powerful single tool in our assurance toolbox.

The Galois high assurance methodology relies on being able to decompose claims about systems to sub-claims about sub-systems, evaluating evidence to track the residual risk. All formal methods use the following procedure to generate evidence that a system satisfies a claim:

1. Construct a mathematical model  $M$  of the system.
2. Translate the claim to a mathematical property  $P$ .
3. Prove that the model  $M$  satisfies the property  $P$ .

Supposing that this is achieved, there are three corresponding sources of residual risk:

1. The mathematical model  $M$  might not be a faithful or complete representation of the system.
2. The mathematical property  $P$  might not be a faithful or complete representation of the claim.

---

<sup>8</sup>It has been said that “all models are wrong, but some models are useful.”

<sup>9</sup>Taking Mark Twain’s advice to “Put all your eggs in one basket, and watch that basket!”

3. The proof that the model  $M$  satisfies the property  $P$  might be faulty.

What makes formal methods so attractive as a high assurance methodology is in certain circumstances it appears that this residual risk can be driven almost to zero: if the system under examination is implemented in software, then a faithful mathematical model of system behavior can be constructed by making use of the semantics of the programming language; claims are often mathematical properties of system behavior; and the formal methods research community has a great deal of experience building reliable tools that can check mechanized versions of mathematical proofs.

However, there are both theoretical and practical obstacles to using formal methods to completely eliminate residual risk. On the theoretical side, Fetzer [9] noted that formal methods can only prove properties of mathematical objects, but all computer systems must run in the real world where they are subject to an array of real-world risks (including bits flipped by cosmic rays) that can never be completely eliminated. This *category mistake* was the source of the controversy surrounding the VIPER processor: parts of which had been formally verified, but even so there could be no guarantees that the physical chip would operate without error [11]. On the practical side, the level of human effort required to formally verify properties is often infeasible for realistic system models. Precisely defining the semantics of real-world programming languages such as C is a hard problem in itself, and applying them to real programs to construct mechanical proofs is often both too time-consuming and overwhelms proof checking tools.

With these caveats in mind, there are many situations where it is practically useful to apply formal methods to generate evidence of claims about a system. A straightforward example is static type checking for modern programming languages such as Haskell. The type system is powerful enough to represent complex program properties (e.g., the state of a network protocol [14]), and the type checker runs over the program at compile time and produces a guarantee that the program properties expressed by types will always hold at run-time. This application of formal methods is fully automatic, scales up to programs of any size, and is robust to catching errors at any project phase from development to maintenance. As such it is an effective technique for generating formal evidence for claims that support an assurance case, and this results in a reduction of residual risk at little cost. In the following sections we will give a brief overview of other formal methods tools that offer good value in terms of high assurance evidence for effort.

### 3.2.1 Static Analysis

Static analysis tools work by examining program text, and are thus complementary to testing regimes that check the dynamic behavior of programs. Static analysis tools check a wide variety of properties, from simple type checking extensions through temporal properties to full correctness, and typically work fully automatically. The properties to be checked are either hard-wired or come from source code assertions, supporting the use of static analysis tools for regression testing during project development and subsequent maintenance. To illustrate the state of the art in static analysis, we will look at two techniques: abstract interpretation and model checking.

Abstract interpretation is a useful technique for finding memory corruption problems such as dereferencing null pointers and buffer overflows. Memory safety of a program is often a critical part of its assurance case, and the evidence produced by an abstract interpretation tool reduces this risk. In keeping with the high assurance methodology, the amount of reduction is determined by the assurance of this evidence, and it must be noted that many abstract interpretation tools available on the market today do not give much indication when they make internal soundness compromises for the sake of efficiency. As to scope, the abstract interpretation tool Coverity Prevent is routinely run over open source codebases up to a million lines of code, and other abstract interpretation tools report similar reach.

Model checking is a technique for checking temporal properties of programs. For example, the SLAM tool developed at Microsoft checks that Windows device drivers conform with system call protocols when they interact with the operating system. SLAM can effectively find errors such as failure to acquire locks before releasing them in device drivers that consist of tens of thousands of lines of C code. The model checking techniques in the SLAM project were built upon at Microsoft Research to develop the TERMINATOR tool, which discovers infinite loops in device drivers that cause the operating system to hang [8]. Like memory safety, temporal properties such as protocol conformance and progress are often critical claims in assurance cases, and model checkers generate formal evidence to support these claims.

### 3.2.2 Verifying Compilers

Verifying compilers are an emerging formal methods technique which have the potential to radically reduce the level of effort required to produce an

assurance case. To achieve an acceptably low level of risk that the evaluated code is the executed code, evaluators often need to look at the target code that is generated by a compiler. However, if the compiler can also produce high assurance evidence that the target code had the same behavior as the source code, then the evaluator could instead examine the source code, which is usually shorter and often contains programmer comments.

The CompCert academic project has developed an optimizing C compiler that produces a formal proof that the PowerPC object code it generates is equivalent to the input C source code. In the same spirit, Galois has developed a VHDL compiler for Cryptol, a domain-specific language for cryptography, which also generates a proof that the VHDL target code is equivalent to the Cryptol source code. Using this compiler a reference implementation of a cryptographic algorithm in Cryptol can be verified to be equivalent to a hardware implementation optimized for performance, reducing the level of effort required to build an assurance case with low residual risk.

### 3.3 Program Understanding Tools

In current practice, manual source code inspection is one component of software evaluation. Because of the computational and mental expense of formal methods on large projects, and the troubles inherent in dealing with legacy software, not designed with the goal of producing assurance evidence in mind, manual inspection is likely to be a necessary part of evaluation for some time to come. Manual inspection cannot eliminate the risk that a software component does not operate as intended, however. The limits of human capacity means that a certain degree of residual risk will remain, proportional to the gap between what we can directly observe in a program's source code and the claims we want to make about its operation.

A key goal of program understanding tools is to help relieve evaluators of the tedious and error-prone extraction of low-level details from source code, allowing them to focus on producing evidence for the higher-level claims that benefit most from human insight, reducing the residual risk in the system.

Typically, time constraints mean that evaluators can only hope to inspect some tiny fraction of the total source code of a large project. And, as projects grow even larger, it becomes more difficult to understand the structure of an entire system, reducing the value of what little code inspection is possible. It is therefore important that evaluators can both determine what components of a system are most critical to the claims they wish to make, and that they

can inspect those components as efficiently as possible to extract evidence to support those claims.

To understand how components of a program interact, evaluators typically need to switch back and forth between different locations in its source files, slowly developing a mental model of what functions can be called from a given location, what operations can affect the value of a given variable, and where the value of a variable may flow once it is set.

All of these tasks can, however, be more easily, quickly, and reliably performed by a computer, using static analysis techniques such as those mentioned in the previous section. When a computer can perform the routine and tedious bookkeeping, the human user is left with the more difficult but more interesting tasks, such as assembling evidence to support the claim of adherence to a high-level security policy.

Thus, program understanding tools aim to make it faster and easier to build a mental model of a large piece of software. They accomplish this by performing a wide variety of simple static analyses that can help answer common questions that occur when trying to comprehend the operation of a software system. At the simple end, they can track definitions and uses to help their users quickly navigate between related sections of source code, such as a function call site and a function definition, or a variable use and its declaration [1, 2].

More complex analyses are also possible, and can further aid in deep understanding. Well-known program analyses can, for example, calculate data flow throughout a program, to determine what other variables the value of a given variable can affect, what other variables can affect its value, and (an approximation of) its possible range of values [10, 7]. All of these pieces of information provide strong evidence for deeper and more sophisticated claims to be made by a human user.

Along with their analysis capabilities, good program understanding tools present the relevant results to their user as seamlessly as possible, with a strong regard for context. Because the overall task they assist is manual source code inspection, good tools can overlay relevant evidence on top of the source code it refers to. To aid in the understanding of high-level architecture, they can also display complete or partial diagrams of component interaction, such as call graphs, execution traces, slices (excerpts of the code that affect or are affected by particular variables), and dependency graphs [6].

Program understanding tools are essentially human interfaces to static analysis tools. They allow users to clearly comprehend the results of analysis

through a high-bandwidth interface, instead of a simple yes/no conclusion. Although a conclusive yes/no result would be ideal, the sound result in many cases is simply “unknown”, based on the information available to the analysis tools, and limits on their computational power. In these cases, it can be useful for the tools to present what information they have determined to the human user, to give the user a head start on determining whether any high-level claims are satisfied.

In summary, program understanding tools help to efficiently bridge the gap between what we can directly observe in the source code of software components and the claims we want to make about an entire system. They accomplish this by inferring and presenting relevant evidence, helping their user become convinced that this evidence supports the desired claims, or helping point out where the evidence contradicts those claims. By reducing the distance between the system and the claims we wish to make about it, they reduce residual risk.

## 4 Summary

Creating a science of security to support high assurance development is all about producing models in which the assumptions are transparent, the methodology is repeatable, and the outcome is an assessment of residual risk. In this paper we have presented a high assurance methodology in which diverse development teams can communicate in a common language to tackle the challenges of developing secure systems. Furthermore, this framework will support formal inference techniques (in particular, in a trust relationship analysis), so that we can use automated reasoning to deal with scalability issues.

Perhaps most importantly, an evidence-based approach lets us tailor the tools that we bring to bear on each claim: formal methods, testing, configuration management, and so forth all have their place in an assurance argument. In the end, it’s all about deploying systems where the residual risk has been minimized, given finite resources and time. Understanding this and managing it effectively is what the science of security is all about.

## References

- [1] Exuberant Ctags. <http://ctags.sourceforge.net/>.
- [2] LXR. <http://lxr.linux.no/>.
- [3] *Summary of a Workshop on Software Certification and Dependability*. The National Academies Press, 2004.
- [4] R. Anderson. Why information security is hard - an economic perspective. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, pages 358–365, 2001.
- [5] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 1 edition, January 2001.
- [6] Sarita Bassil and Rudolf K. Keller. Software visualization tools: Survey and analysis. In *Proceedings of the International Workshop on Program Comprehension*, pages 7–17, 2001.
- [7] CEA-LIST and INRIA-Saclay. Frama-C. <http://frama-c.cea.fr/>.
- [8] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In Thomas Ball and Robert B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418. Springer, August 2006.
- [9] James H. Fetzer. Program verification: the very idea. *Communications of the ACM*, 31(9):1048–1063, 1988.
- [10] GrammaTech. CodeSurfer. <http://www.grammatech.com/products/codesurfer/>.
- [11] Donald MacKenzie. *Mechanizing proof: computing, risk, and trust*. MIT Press, 2001.
- [12] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Software Security Series. Addison-Wesley, February 2006.

- [13] L. Northrup, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon, 2006. The 2006 report for a 12-month study of ultra-large-scale systems software, sponsored by the United States Department of Defense.
- [14] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In Andy Gill, editor, *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, September 2008.
- [15] Fred Schneider. *Trust in Cyberspace*. National Academy Press, 1999.
- [16] Stephen Toulmin. *The Uses of Argument*. Cambridge University Press, 1958.