Techniques for Trusted Software Engineering

Premkumar T. Devanbu
Dept of Computer Science,
University of California,
Davis, CA 95616 USA
+1-530-752-7324
devanbu@cs.ucdavis.edu

Philip W-L Fong
School of Computing Science
Simon Fraser University
Burnaby, Canada V5A 1S6
+1-604-291-4277
pwfong@cs.sfu.ca

Stuart G. Stubblebine
AT&T Laboratories – Research
180 Park Ave,
Florham Park, NJ07932, USA
+1-973-360-8354
stubblebine@research.att.com

ABSTRACT
How do we decide if it is safe to run a given piece of software on our machine? Software used to arrive in shrink-wrapped packages from known vendors. But increasingly, software of unknown provenance arrives over the internet as applets or agents. Running such software risks serious harm to the hosting machine. Risks include serious damage to the system and loss of private information. Decisions about hosting such software are preferably made with good knowledge of the *software product* itself, and of the *software process* used to build it. We use the term *Trusted Software Engineering* to describe tools and techniques for constructing safe software artifacts in a manner designed to inspire trust in potential hosts. Existing approaches have considered issues such as schedule, cost and efficiency; we argue that the traditionally software engineering issues of configuration management and intellectual property protection are also of vital concern. Existing approaches (e.g., Java) to this problem have used static type checking, run-time environments, formal proofs and/or cryptographic signatures; we propose the use of trusted hardware in combination with a key management infrastructure as an additional, complementary technique for trusted software engineering, which offers some attractive features.

KEYWORDS
Safety, security, mobile code, cryptography, analysis, verification.

1 INTRODUCTION

Installing new software on a machine is risky. Poor quality or malicious software can do serious harm. The traditional defense has been to install only high-quality software products from well-known vendors.

This method is not always applicable: companies such as AT&T provide world-wide web (WWW) hosting services. Web content (web pages, and associated software such as common gateway interface, or CGIs) is hosted on fast, reliable servers on behalf of other companies or individuals. Since CGIs are ordinary applications, they can damage the hosting company's ability to provide non-stop service. Thus, hosting companies need ways of developing confidence that the CGIs have certain safety properties (e.g., they don't delete files, write to operating system tables, use up too much CPU time/Memory, etc.). The traditional model also breaks down in the context of technologies such as Java[15, 17], particularly with applets and mobile code. The simple act of browsing a web page can cause software to be installed and run on a hosting machine.

When offered software of unknown provenance to be installed and run on a hosting machine, the host's ($\mathcal{H}$) decision would best be based on *reliable evidence* concerning:

1. *The software process*: how was the software built? what were the design, development and testing practices used?
2. *The software product*: What are the properties of the software itself?

We use the term *trusted software engineering* to describe tools and techniques that can be used to construct safe[1] software that inspires trust in hosts. Engineering concerns such as cost, efficiency, delay, etc., are of vital importance; in addition, the vendor ($\mathcal{V}$) can be expected to be deeply concerned about disclosure of valuable intellectual property.

In an earlier paper[11] we explored techniques for the *process* side of trusted software engineering: the concern there was to find ways in which $\mathcal{V}$ could convince (quickly, and at low cost) a host ($\mathcal{H}$) that $\mathcal{V}$'s testing practices were rigorous, without disclosing too much information. In this paper, we turn to the *product* side

[1] We deliberately refrain from defining *safety* in this paper. Different hosts may have different safety policies; for generality, our approach remains agnostic on the precise nature of safety. However, we discuss specific possible applications.

0-8186-8368-6/98 $10.00

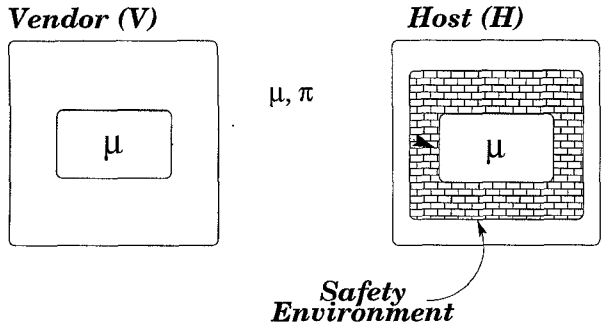

Figure 1: General architecture for mobile code safety

of trusted software engineering; how can $\mathcal{V}$ convince $\mathcal{H}$ that a software product has certain desired safety properties?

The outline of this paper is as follows: we begin with a description of the design parameters of concern to trusted software engineering; we then use these parameters to analyze existing approaches to this problem. After describing trusted hardware systems, we explore the role they could play in trusted software engineering. The paper concludes after a description of implementation considerations and possible difficulties with this approach. An outline of the ideas discussed in this paper were presented earlier in a position paper [10]. In this paper, we describe full details including a key management infrastructure, details of our approaches to dealing with resource limitation, and describe an implementation for Java [15] bytecode verification.

2 APPROACHES TO SAFETY

In this section, we describe some existing approaches to safety and trust in software. Figure 1 represents current approaches to safety. There is a vendor, $\mathcal{V}$, who produces a piece of code (perhaps mobile code) μ. This μ gets shipped to a host $\mathcal{H}$, along with another artifact π, which may be claims about properties of μ and/or proofs of such claims, and/or a cryptographic digest of the software. The host $\mathcal{H}$, upon receipt of μ and π, may conduct an analysis of μ and π, to evaluate the claims (if any) that $\mathcal{V}$ made about μ, and to determine if μ can be trusted to run safely on $\mathcal{H}$'s machine. After such analysis, $\mathcal{H}$ may elect to run μ, (often) within some run-time environment. The safety analysis process, and the run-time environment are together represented as an enclosing "brick wall" denoted as a *safety environment* in figure 1.

When considering such approaches, which enable hosts $\mathcal{H}$ systematically develop confidence in the safety of system μ built by $\mathcal{V}$ (and then run μ), there are several important criteria to be considered, from the perspective of both $\mathcal{V}$ and $\mathcal{H}$:

1. *Cost:* how much additional skilled personnel time and/or process interval is involved?
2. *Performance:* what is the run-time overhead?
3. *Disclosure:* how much intellectual property disclosure is involved?
4. *Configuration Management:* when the inevitable weaknesses are discovered in the software infrastructure that enforces security and safety, how easy is it to distribute upgrades? Likewise, can one protect customers using "outdated" client software with known security vulnerabilities?
5. *Security:* what is the nature of the security guarantee provided? Is it formally proven, informally established (perhaps by a social review process) or is there no guarantee? (more risky)?

We will argue that cost, performance, and security considerations have been of paramount consideration in current approaches to this problem; we will further argue that the traditionally software engineering concerns of configuration management and disclosure are also vital. We suggest a complementary approach to address these issues.

It is difficult or impossible to optimize all these criteria simultaneously. Thus, one way to achieve perfect, formal security is to ask the vendor to provide fully annotated source code. $\mathcal{H}$ then creates a complete proof based on the annotated source, then compiles the source code, and allows it to run. This gains formal assurance at great cost, while also demanding high disclosure of the vendor. Another approach is for $\mathcal{H}$ could run the software on a "sandbox" simulator which analyzes and predicts the effect of each step before allowing it to execute. This approach offers high security, low personnel cost and low disclosure but at high run-time cost; also this approach requires every $\mathcal{H}$ (there may be millions) to upgrade the simulator if it is found to have a security hole. Finally, the trivial approach of letting any program run offers no security, but maximizes everything else. The challenge for trusted software engineering is to construct a design with a right combination of features for a given application. Our goal in this paper is to describe a new technique, involving trusted hardware and key management, that can lead to better trusted software engineering solutions for some applications.

We describe and evaluate current approaches to establishing the safety of software products, two of which have their roots in the WWW. These are: *Java*, *ActiveX*[1, 13] and *Proof Carrying Code (PCC)*[19]. Each approach makes different tradeoff; we now discuss them in detail.

2.1 JAVA

Java is a strongly-typed, object-oriented language[15], which in combination with a well defined run-time environment[16] provides a safe environment for hosting mobile code. Java source language programs are compiled into Java bytecodes, an equivalent binary representation, which are interpreted by the Java Virtual Machine (JVM). WWW Browsers such as Netscape^{TM} include a JVM that can execute Java bytecode programs (applets) embedded in web pages; accessing such web pages causes the JVM in the browser to execute the applets. Applets are downloaded and executed in an "almost" transparent manner; safety of applets is thus a critical aspect of Java.

Type safety is at the heart of the Java safety model. Before running, every Java applet is typechecked statically; if a program passes this typecheck, there is a reasonable belief (as yet formally unverified, although efforts are underway) that there will be no type confusion, *i.e.*, it is impossible for a variable to change its type at runtime. Avoiding type confusion is critical(for details, see [17]). Java source language programs are type checked by the Java compiler prior to being compiled into bytecode applets. Since browsers execute applets received from untrusted web servers, the associated JVMs have to recheck the applets for type safety prior to executing them. This fairly complex process, called *byte code verification*, adds to the overhead of executing an applet. To allow for this process, Java bytecodes must have enough information to allow type checking[2]. Furthermore the byte code verification process is embedded (via the JVM) in web browsers; when faults or weaknesses are discovered in the bytecode verification process, *every web user has to download a new copy of the browser.* Many of the millions of web users are unlikely to upgrade their browsers, thus leaving themselves vulnerable to hostile acts by malicious applets.

With reference to figure 1: mobile code μ consists of Java bytecodes. The vendor makes no claims about μ, i.e., there is no π—however, the desired property is type safety, which is checked by the byte code verifier. The safety environment at the host's site consists of the byte code verifier and the Java virtual machine, with the associated security managers. According to our five criteria (Section 2 above): the Java model has (1) no additional programmer cost; (2) a significant run-time overhead for bytecode verification, and for the "sand box" (3) significant disclosure (the source code of the applet) and (4) a substantial upgrade problem. The security guarantee (5) provided here is "somewhat" formal; the procedures are described in great detail in documents subject to rigorous public review; efforts are underway to put them on a formal basis. Since upgrade decisions for the host-side software are made locally, the control of the security policy is distributed. This can lead to vulnerabilities.

[2]In fact, it has been demonstrated that Java source codes can be reconstructed from byte codes.

2.2 ACTIVEX

The ActiveX model is similar to Java: application code embedded in web pages is downloaded and executed when the pages are visited. However, the embedded applications are in the form of *binaries*, which are executed on the "bare" machine. This lets embedded applications from web pages run with the same privileges as regular applications The ActiveX model calls for checking that the embedded application is signed by a "known and trusted" party. The security and safety of the application is left unspecified. This is potentially risky. To quote the Princeton Safe Internet Programming group[13]:

> "ActiveX security relies entirely on human judgment. ActiveX programs come with digital signatures from the author of the program and anybody else who chooses to endorse the program."

With reference to figure 1: mobile code μ here is just the ActiveX binary; π is the cryptographic signature on μ by a trusted agent. The safety environment is just the checking of the signature π against μ. According to our criteria, this approach (1) involves very little additional programmer work (2) involves very little run-time overhead, (3) involves no disclosure beyond the binary and (4) will probably not need upgrades, since the signature checking software is simple, well-understood and quite stable. However, since the approach offers no explicit guarantees of security (5) besides the wisdom and good intentions of the party which signed the embedded application, it is fraught with risk. There is another important limitation; although there are a considerable number of software vendors, few of these vendors may have brand name recognition among consumers, and thus enjoy broad acceptance in this trust model. However, in general, consumers would suffer from being limited to using code signed by large, well-known vendors.

2.3 PROOF CARRYING CODE

A powerful approach to establishing safety (or other) properties of programs is through *formal verification.* Formal verification typically involves 3 steps: first, annotate several program points with invariant assertions (typically at the start of loops) that hold when those points are exercised. Second, use a verification condition generator uses the assertions and the semantics of the program to generate a verification condition. Finally, produce a proof (usually by hand) that establishes that the verification condition is true given some initial conditions. Usually, the verification condition relates to

the desired safety property. Clearly, it would be impractical for a host $\mathcal{H}$ to formally prove safety for all received programs.

Necula[19] has proposed an elegant approach to code safety using formal verification. His work relies on the fact that proof *checking* is much simpler and faster than proof *creation*. In his framework, code vendors enhance binary programs with invariant assertions and package them together with a safety proof. This entire bundle is a *proof carrying code* ($\mathcal{PCC}$). Upon reception, the host $\mathcal{H}$ processes the assertions and the instructions in the binary to yield a verification condition. When the enclosed proof of the verification condition has been checked by $\mathcal{H}$, the program can be run at binary speeds!

With reference to figure 1: mobile code μ consists of the (annotated) binary, and π is the proof. The safety environment consists of the verification condition generator, and the proof checker. According to our five criteria: this approach can be expected to involve a large amount (1) of programmer time, since proofs must typically be created by hand[3]. Run-time overhead (2) is significant: for a (roughly) 1 Kbyte program, the proof checking takes about 2 ms for published examples [19]. However, in general, proofs could be very long, which would result in increased checking time (and transmission time). Finally, depending on the particular proof, a lot (3) can be disclosed: the invariant assertions and the proof may reveal a lot about the program. For example, to establish type-safe pointer generation, the memory layout of all data structures must be disclosed. Since each host $\mathcal{H}$ has a copy of the verification condition generator and the proof checker, it will be necessary to do a large number of upgrades (4) should it be determined to be faulty. The greatest strength of this approach (5) is that it provides a precise, unforgeable, irrefutable formal characterization of the safety of the mobile code. The host side proof checker is configured locally; so the security environment administration is distributed.

2.4 ANALYSIS

The approaches discussed in this section all attempt to achieve trusted software engineering in different ways. Each approach leads to a particular level of cost, performance, disclosure, release management, and security. Each approach also makes a choice in a design space whose dimensions are the currently available technical options (such as public key cryptography, strong typing, run-time checks, and formal proofs and proof checking). In this paper, we advocate other technologies for this design space: *trusted hardware*, and configuration management by *key management*.

[3]However, Necula is at work on compilers that can generate proofs of certain kinds of properties

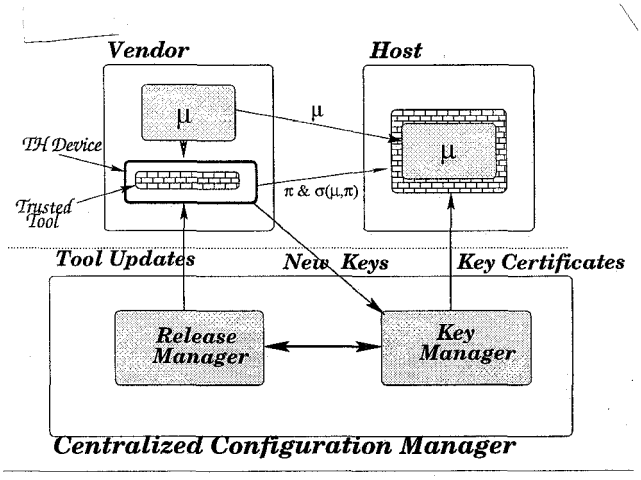

Figure 2: An alternate architecture for mobile code safety

3 A COMPLEMENTARY ARCHITECTURE

We propose the use of *trusted hardware* ($\mathcal{TH}$), in conjunction with centralized *configuration management* as an additional technique for trusted software engineering. In particular, we propose that configuration management be handled by the distribution of key credentials (*revocations* and *certificates*) within a public key system. We assume some basic knowledge of cryptography: readers unfamiliar with cryptographic terminology are referred to the Appendix(page 10) for details.

The proposed architecture is shown in Figure 2. There are two major changes from Figure 1. First, part of the safety environment is moved from the $\mathcal{H}$'s machine to the $\mathcal{V}$'s machine; this removal is represented as a "thinner wall" at the host's machine. At the vendor's site, this piece of $\mathcal{H}$'s safety environment is ensconced as a trusted tool (shown as a "slice of brick wall") inside a trusted hardware device (described below). Software (μ) produced by $\mathcal{V}$ is processed by the trusted tool, resulting in some analysis results (π). The software and the analysis results together are cryptographically signed ($\sigma(\mu, \pi)$) by a key embedded in the trusted hardware device.

Now, μ, π and $\sigma(\mu, \pi)$ are shipped to the host $\mathcal{H}$, and used by the (diminished) safety environment on $\mathcal{H}$'s machine. The $\mathcal{H}$ can use the signature in lieu of certain computations that it would otherwise have to perform (details are given below). We also posit (below the dotted line) a configuration management system, which serves two logical functions: 1) updating the trusted software tool that runs at the vendor's site within the trusted hardware device, and the associated private keys, 2) updating certificates for the keys used at the host's site for signature verification.

3.1 TRUSTED HARDWARE

Several manufacturers offer physically secure co-processors in PCMCIA [5, 2] and PCI [3] form factors. These devices contain a CPU, volatile and non-volatile memory, built-in cryptographic[18] facilities (symmetric & public key algorithms, random number generation, etc), private keys, and certificates. The programs and non-volatile data contained in such $\mathcal{TH}$ devices are physically protected: attempts to access or modify them will render the device non-functional. Physical security is a critical requirement in the intended application of such devices (highly security-critical financial and defense uses) and is regulated by national and international standards [6]. Since these are general purpose machines, one can conduct arbitrary computations on them, and generate outputs signed with a secret private key. This allows for software tools, embedded in $\mathcal{TH}$ devices, to be run by an untrusted $\mathcal{V}$ a manner that can inspire trust in a skeptical $\mathcal{H}$, based on a signature. Of course, this trust is subject to cryptanalytic assumptions such as the difficulty of forging signatures, and the evolving technology of physical security.

The crucial observation is that $\mathcal{TH}$ devices, in conjunction with a centralized, trusted configuration management system, can support *trusted, distributed enactment of software engineering processes*, while offering cost, performance, disclosure, upgrade management and security advantages. Trusted tools in $\mathcal{TH}$ devices can be distributed to untrusted developers; software created (and signed) by this device can be configured and hosted as if the tools were run locally. Since computations related to security and optimization can be safely performed remotely, *cost* and *performance* benefits can be obtained. A simple cryptographic signature can carry the weight of a lot of information; so $\mathcal{V}$ may not need to *disclose* as much. Additionally, *configuration* management could be simplified, since there are typically far fewer vendors than hosts; from the hosts' point of view, release upgrade is done by key management.

3.2 SOFTWARE CONFIGURATION USING KEY MANAGEMENT TECHNIQUES

Key management is used in concert with $\mathcal{TH}$ devices to a) take defective (or outdated) $\mathcal{TH}$ devices out of service, b) provide fixes for host-side security software with known weaknesses and c) ameliorate the risk of compromise of hardware devices. In this section, we discuss a) and b); c) is discussed later (Section 6.1). In the discussion in the section below, we assume that some verification software, *which would normally be run by the host* $\mathcal{H}$, is now running at the vendor's site within a $\mathcal{TH}$ device. For brevity, we also assume a centralized simple configuration manager, denoted ω.

We now discuss a), the revocation of faulty versions of $\mathcal{TH}$ devices. As an example, assume that some code C has been successfully verified by a $\mathcal{TH}$ device with tool T_1 (with private key K_1^{-1}), which attests to this fact with a signature $\sigma_{K_1^{-1}}(C, \mathcal{D})$. $\mathcal{D}$ describes the verification or analysis performed, and other details such as the version of the analysis software, and the vendor's id. $C, \mathcal{D}$ and $\sigma_{K_1^{-1}}(C, \mathcal{D})$ are sent to a host $\mathcal{H}$. The public key K_1 (for K_1^{-1}) is introduced to $\mathcal{H}$ via a certificate $Cert_\omega(K_1)$ from the configuration manager ω. $\mathcal{H}$ (assumed to know K_ω already) extracts the key K_1 from the certificate, checks the signature $\sigma_{K_1^{-1}}(C, \mathcal{D})$ against C, allows C to run if $\mathcal{D}$ satisfy the policy for executing such code. These certificates do not have to be distributed with the code C in it; they could be distributed independently, using push technologies such as a Marimba [4] channel.

Now, suppose, that a software bug is discovered in the current version of the verifier software. Rather than distributing new versions of the software to every host, the ω effectively revokes the validity of the current version of the verifier software by assuming that hosts authenticate software subject to recent-secure authentication policies [20]. That is, if the hosts require recent statements concerning the authenticity of software configurations and $\mathcal{TH}$, then we can assure bounded delays for fail-safe revocation of vulnerable configurations. For such vulnerable configurations, ω stops issuing timestamped certificates attesting to the validity of verification software indicated in $\mathcal{D}$. Consequently, distributed entities are unable to obtain fresh statements vouching for the validity of the verifier software. This has the effect of hosts being unable to satisfy their recency policy on checking the authenticity of verification software indicated in $\mathcal{D}$. Consequently, the host treats the signed code as suspect. This general approach for using a trusted-third party revocation service and recent-secure authentication is first described in [20].

The vendor in possession of the $\mathcal{TH}$ device is then alerted to obtain a new version T_2. In figure 2 the release of T_2 would be handled by the configuration manager. Reconfiguring $\mathcal{TH}$ for new software consists of the $\mathcal{TH}$ authenticating a new software upgrade from the configuration manager using a public key of the configuration manager stored on the card. Note, this procedure does not require encrypted messages. Hence, the vendor has some assurance that details concerning testing are not leaked. Future signatures prescribe the use of the new verification software in $\mathcal{D}$.

We now turn to b), fixing old versions of host-side software with new $\mathcal{TH}$ devices. Assume as before that T_1 is the current version of the software verifier in a $\mathcal{TH}$ device. Also assume some host-side security software H_1, which is found to have a weakness. The current

approach is to require all hosts to download a new version, H_2, which plugs the security hole. With $\mathcal{TH}$ devices, another approach is possible in some cases: a new (branch) version of the $\mathcal{TH}$, $T_{1\ 1}$ is issued, which plugs the security hole in H_1. All hosts running H_1 are told to accept only software processed by $T_{1\ 1}$ verifiers. This is accomplished by revoking and issuing the appropriate certificates associated with the current valid version of the verification software indicated in $\mathcal{D}$.

The functionality at the host side remains identical; this reduces or simplifies the host's side administration. This approach both saves time for the hosts and provides additional security by centralizing the configuration management of the security infrastructure. However, this may not be a perfect solution; plugging the host's security "hole" may require very strong verification which may reduce functionality.

Other configuration management strategies can be used in conjunction with more complex host-side safety requirements. In combination with a flexible trust policy management infrastructure [9], this approach offers a high degree of flexibility. In the most general case, one can envision a situation where the host $\mathcal{H}$ specifies a set of safety requirements, and describes the configuration (version information) of his safety environment. With this knowledge, the centralized configuration management system can automatically distribute certificates to $\mathcal{H}$; these certificates ensure that the right combination of host-side and vendor-side safety environment software is in effect, for the specific safety policy required by $\mathcal{H}$.

We now discuss the application of our approach to Java and $\mathcal{PCC}$.

3.3 JAVA AND $\mathcal{TH}$

First, consider Java. The bytecode verifier is ensconced in a $\mathcal{TH}$, and made available to Java developers. When a developer is finished developing an applet (using conventional tools on $\mathcal{H}$), she submits the bytecode for the applet to $\mathcal{TH}$. The built-in bytecode verifier in the $\mathcal{TH}$ verifies it, and if the verification is successful, the $\mathcal{TH}$ outputs 1) a signature σ, using a private key $K_{\mathcal{TH}}$, specific to that $\mathcal{TH}$. $K_{\mathcal{TH}}$ is extracted from C. A receiving browser can believe that the applet's bytecode has been verified (because of σ) by a trusted bytecode verifier. There are some complications here; an applet may be composed of several distinct class (`.class`) or archive (`.jar`) files. Our approach calls for each of these files to be verified and signed separately. Our approach to the resulting complications are described in Section 5.

Now, suppose a flaw is discovered in the bytecode verification algorithm, or the implementation in the version r installed in $\mathcal{TH}$ devices. The configuration manager ω of the $\mathcal{TH}$ issues a new revision of the software r', and arranges to upgrade all holders of $\mathcal{TH}$ devices with the new software. Simultaneously, ω sends out certificates, revoking the keys of $\mathcal{TH}$ devices with r versions, and issuing new keys for devices with r' versions. Additionally, if a fault is discovered in the Java (host-side) run-time environment, which could be fixed by a modified byte code verifier, a similar strategy could be undertaken.

This approach offers clear improvements for two of the five criteria, in the case of Java: *upgrades* and *security*. There are also *performance* advantages in some cases. First, in the case of Java, there is no additional work for the programmer, apart from submitting the bytecodes to the $\mathcal{TH}$ for verification. Second, the receiving browser doesn't have to typecheck the bytecodes; this could speed the applet, thus improving the user experience of the web page. Third, resource limited computers with embedded JVMs will not have the resources to run bytecode verifiers and would benefit from our approach. Finally, for the browser user, security flaws in the bytecode verifier such as one discovered recently by the Kimera researchers [21] no longer necessitate downloading an entire new version of the browser: release management becomes a matter of key management! Rather than the vast number of web users updating their browsers, we can have a far more manageable number of updates for the applet developers. Furthermore, by automating the key management on the browser side (using *push* or *pull* technologies) we can make it transparent to the browser users; this will increase security for users unaware of security flaws. By the same token, certain weaknesses in the Java virtual machine itself, such as one that allowed the creation of rogue classloaders ([17], pp 77-82) can be fixed by distributing more restrictive versions of the bytecode verifier to vendors, and doing the appropriate key management on the host side.

Since Java bytecodes are essentially source code [22], applets contain all the information available in source code. One defense is to use bytecode obfuscators; our approach is compatible this defense. In addition, Java bytecode recompilers [23] (which produce "fat", multiplatform binaries that contain binary code in addition to or instead of bytecodes) can also be accommodated. In this case, the $\mathcal{TH}$ device is provided the original bytecodes, the compiled binary, and a "proof" (consisting of the sequence of meaning-preserving rewrites used to produce the "fat" binary") that the binary corresponds to the bytecodes. The $\mathcal{TH}$ device checks the bytecodes for type correctness, and then ensures the sequence of rewrites provided in the "proof" are known to be meaning preserving, and that executed correctly, they produce the given "fat" binary. The result can then be signed as equivalent to statically typechecked Java byte-

code. The vendor only need disclose the binary and the signature; the Java bytecode can be protected. This approach provides both disclosure and performance benefits; Without bytecode verification or compilation, the host can run at binary speeds.

3.4 $\mathcal{PCC}$ AND $\mathcal{TH}$

If the code vendor is creating $\mathcal{PCC}$, the $\mathcal{TH}$ approach offers even greater benefits. In this case, we embed a verification condition generator, and a proof checker in the $\mathcal{TH}$ device. The scenario is as follows: $\mathcal{V}$ creates the binary, the invariant assertions, and the safety proof, and submits the lot (as a $\mathcal{PCC}$) to the $\mathcal{TH}$ device at his site. The $\mathcal{TH}$ analyzes the binary and assertions, regenerates the verification condition, and checks the proof. If the proof checks out, the $\mathcal{TH}$ outputs a signature signing (just) the binary, and the verification condition. $\mathcal{V}$ now makes a package consisting of the (unannotated) binary, the verification condition, and the signature. The $\mathcal{H}$, upon receiving this signed package, checks the signature against the binary and the verification condition. This gives him confidence that a trusted party in the $\mathcal{TH}$ has checked the proof of the verification condition. As long as the verified condition subsumes his safety policy, $\mathcal{H}$ can boldly run the binary.

This approach offers significant advantages on performance, disclosure, and upgrades. Since the proof is checked at the producer's site, there is no run-time overhead at the $\mathcal{H}$'s site. Secondly, the only thing that leaves the producer's site is a signature; the assertions and the proof do not have to be disclosed. Finally, configuration management is handled automatically via key management.

4 Implementation: Trusted, Resource Limited Computing

Price, compatibility, heat dissipation difficulties and physical security considerations force extremely tight engineering constraints on the design of $\mathcal{TH}$ devices, specially in the PCMCIA format. The computing resources available, particularly memory, on these cards is limited. As technology evolves, resources in $\mathcal{TH}$ devices will probably always be several orders of magnitude below what is available on a current conventional computer. On the other hand, $\mathcal{TH}$ devices are always used along with an (untrusted) conventional machine. For this reason, it is natural that trusted software engineering tools not run purely on the $\mathcal{TH}$, but as a distributed computation involving the hosting computer. However, the hosting machine $\mathcal{P}$ is under the control of an untrusted party, and any supporting computations performed by $\mathcal{P}$ are subject to tampering. To deal with this, we have adopted the following posture. The $\mathcal{TH}$ uses the $\mathcal{P}$ as a potentially unlimited computing resource, but always retains a small amount of trusted memory to serve as a "digest" of the operations delegated to $\mathcal{P}$. This digest is used to check the validity of the results returned by $\mathcal{P}$. If tampering by $\mathcal{P}$ is detected, $\mathcal{TH}$ will simply halt the computation and terminate. Thus, we place large data structures such as stacks, queues, and tables in $\mathcal{P}$, and check operations using a small amount of memory in $\mathcal{TH}$.

This approach draws upon memory-checking techniques developed in the theory community [8]; however, those approaches use very strong information theoretic considerations, which allow the $\mathcal{P}$ unlimited computing power to mount an attack on $\mathcal{TH}$. In particular $\mathcal{P}$ can completely simulate $\mathcal{TH}$. Because of these restrictive assumptions, their approaches lead to unattractive implementations. In our case, $\mathcal{TH}$ spends at most a polynomial amount of time on the size of the input, and has access to secrets (keys) unknown to $\mathcal{P}$. Additionally, the adversary, $\mathcal{P}$, enjoys at most constant speedup factor over $\mathcal{TH}$. Under these conditions, $\mathcal{P}$ cannot simulate $\mathcal{TH}$. More efficient implementations of memory-checking protocols are possible, which offer acceptably low probabilities of memory compromise.

A full discussion of this approach and the security of the approach is presented in [12]; we have developed schemes for handling implementations of stacks, queues, and associative arrays implemented as binary trees. For brevity, we only present our implementation of stacks. In Figure 3, the stack is shown just after the push of an item N. There is always a signature of the stack maintained in the $\mathcal{TH}$ device. Prior to executing the push, the signature σ of the stack is in the $\mathcal{TH}$ device; when an item N needs to be pushed on to the stack, $\mathcal{TH}$ computes a new signature σ' as shown in the figure. Then the new item N and the old signature σ are given to the $\mathcal{P}$ stack implementation, with a request to execute a push. The signature σ' is retained in the $\mathcal{TH}$ device's memory as defense against tampering by $\mathcal{P}$. Thus, when a pop command is issued, $\mathcal{P}$ is expected return the top item N, and signature of the rest of stack, σ. Then the original signature σ' is recomputed as shown in the figure and checked against the value stored in $\mathcal{TH}$. It is infeasible for $\mathcal{P}$ to spoof $\mathcal{TH}$ by forging the values of N or σ so long as $\mathcal{TH}$ retains σ'. Thus the stack invariants are preserved. The approach described here uses only a constant number of bits in the $\mathcal{TH}$ device, irrespective of the size of the stack; existing methods use a logarithmic number of bits, which is exactly the information theoretic bound [8]. For our application, with limited adversaries, this is adequate. If information-theoretic security are desired, our security could be increased by using a counter in the $\mathcal{TH}$ device, and inserting signed counts into the stack. Our implementation is also simpler; each stack operation executes in constant time, whereas [8] requires $O(log(stack-size))$ operations (amortized) for

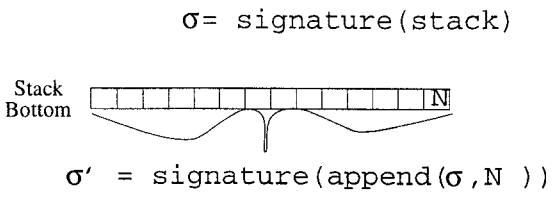

Figure 3: A resource-limited, "secure" implementation of stacks

each stack push and pop.

5 IMPLEMENTATION: JAVA BYTECODE CERTIFICATION

To demonstrate our approach, we are implementing a Java byte code verifier suitable for embedding in a $\mathcal{TH}$ device. Our approach has necessitated a redesign of the Java bytecode verifier. A preliminary implementation of our bytecode verifier can be tested by email (send a "help" message to `genserver@research.att.com`). Upon receipt of Java byte codes, it will verify it, and send the results back by email. This version is under continuous refinement. Our approach of verifying and signing each class file separately creates some special implementation issues, which we now discuss. Recall that in Java, each class file contains the implementation of one Java class.

The verification process, as described in [16] comprises several passes. The first 2 passes ensure that the class file is laid out correctly. The magic number, symbol table entries, instruction sizes and arguments etc. are all checked. All branch statements are examined for target validity. The third pass actually does typechecking. Since bytecodes are not structured programs, this involves control- and data-flow analysis. The final phase includes checks on subroutine invocations and is conducted at run-time. Currently, the bytecode verification process is tightly integrated with the JVM; verification is interleaved with loading, linking and execution. This is necessary: in general, the typesafety of a Java class file can only be established within a global linking context. The typesafety of a statement like `a = b` within a class file (where `a` and `b` are instances of other, different classes) can be established only by loading the corresponding class files. Our approach calls for each class file to be verified and signed separately. Therefore we cannot process a class file *per se* and certify it typesafe with a signature. From each class file C, we create a list of *obligations*, and *commitments*. The obligations list the compatibility relationships that must hold between other classes referred to by C; *commitments* list the relationships between the C class and other classes that are guaranteed by the bytecodes in C. We then sign C together with the commitments and obligations induced by C; the signature certifies both that C has been verified, and the correctness of the linking information. Full details are omitted due to space constraints, and can be found in [14]. A suitably modified JVM can make use of this signed information, avoid verification, and speed up the linking process. This part of the work is still ongoing.

As in [21], we have adopted a "cleanroom" approach to our implementation of the verifier. There is no available formal description of the bytecode verifier; so we have tried to align our implementation closely with the description given in the JVM book [16]. For each part of the JVM description, there is an allied, clearly identities portion of the source code in our implementation. As in the clean room approach, we use statistical testing, with millions of test cases generated by random mutations of legal applets [21]. Testing is underway; after comprehensive testing, embedding in a suitable $\mathcal{TH}$ device will be undertaken.

6 ANALYSIS AND CONCLUSION

In this section, we discuss the problem of physical security compromise, and other approaches to the problem of trusted tools. We also explore the criteria under which this approach is applicable. We conclude with a brief discussion of our future plans.

6.1 HARDWARE COMPROMISE

The security of physical devices and the technology to circumvent protection mechanisms is continually evolving. $\mathcal{TH}$ devices have been compromised [7]. PCMCIA cards and PCI cards (which contain batteries, and can erase secret memory when intrusion is detected) are less vulnerable to attack than smart cards. However, as time evolves, devices once thought secure may become vulnerable. Our goal is to develop an integrated framework that a) allows reasonable recovery from compromise. b) discourages attempts to tamper and c) combines physical integrity of the $\mathcal{TH}$ devices with vendor's identity as a basis for trust.

Recovering from Compromise If a particular $\mathcal{TH}$ is suspected to be compromised, of if tampering is suspected (see following section) its key can be revoked. Also, if the compromise *per se* is undetected, but an unsafe program is discovered to have been signed by a particular $\mathcal{TH}$ device, the key for that device can be revoked.

Discouraging Tampering In general, it is more difficult to physically compromise a $\mathcal{TH}$ device while it is in operation. To this end, we advocate that a $\mathcal{V}$ be required to install his $\mathcal{TH}$ device permanently in a networked machine. After installation (until the $\mathcal{TH}$ device is taken out of service) it will be challenged with the current time at random intervals by an authenticated server. It shall respond with its signature (using its private key) on the challenge data. The mean periodicity can be adjusted to discourage attempts to remove the $\mathcal{TH}$ device and penetrate it off-line. This is a kind of periodic inspection (similar to ones used in arms control surveillance regimens [7]) by electronic means. If challenges are unanswered, the key associated with that $\mathcal{TH}$ device could be revoked via the configuration manager (Figure 2), and the vendor $\mathcal{V}$ may be required to produce the $\mathcal{TH}$ for inspection.

Involving the Vendor To tightly "bind" the vendor $\mathcal{V}$ with a specific $\mathcal{TH}$ device, the software in the $\mathcal{TH}$ device could include information about the vendor's identity. Verified software is packaged along with the vendor's identity and signed with the $\mathcal{TH}$ device's key. The signature establishes that $\mathcal{V}$ owns the $\mathcal{TH}$ device, and is responsible for its integrity. If $\mathcal{V}$ loses the device, or it ceases to function, he is responsible for notifying the configuration manager (See figure 2), which can revoke the device key. If software signed by a certain device $\mathcal{TH}$ is known to be unsafe, this ownership signature provides evidence of ownership of the device. The owner of the device can be called upon to make the device available for inspection.

Upon receipt of this signature, the host $\mathcal{H}$ only has to decide whether she trusts that $\mathcal{V}$ can be relied upon to not compromise the device $\mathcal{TH}$. Notice that the role of $\mathcal{V}$'s identity is quite circumscribed (as compared to ActiveX) in this usage: here, it only means that $\mathcal{V}$ owns, and takes responsibility for the integrity of a particular $\mathcal{TH}$ device. The problem here is that known vendors enjoy an advantage; (though perhaps not as significant as with ActiveX) they are under less suspicion of tampering.

6.2 OTHER APPROACHES TO TRUSTED TOOLS

An alternative approach to trusted hardware is multiparty computation. The idea here is that several mistrusting entities can run the analysis/verification and certify the results. This approach fits tightly into the framework we proposed with an important exception: the code is disclosed to the verification entities. In the case of byte code verification in Java, this is not a problem, since bytecodes are source code [22].

The configuration management technique illustrated in figure 2 would be fully applicable. Keys for old buggy versions of verifiers could be revoked. Likewise, weaknesses in in legacy versions of the browsers could be "plugged" by deploying stricter bytecode verifiers, and doing the appropriate key management.

6.3 APPLICABILITY

$\mathcal{TH}$ devices can offer advantages for performance, configuration management, and disclosure. Performance advantages are obtained by "pre-computing" information in a trusted manner at the vendor's site. Configuration management advantages may also obtain. However, the unique advantage of $\mathcal{TH}$ devices is most vivid in the case where disclosure is a vital concern. The vendor's private information (needed for verification) is kept at the vendor's site; the only information leaving the site is a signature.

6.4 CONCLUSION

We have discussed some novel techniques for trusted software engineering: trusted tools in trusted hardware, and configuration management by key management. We described our progress on implementation work. It is important to emphasize that trusted hardware and key management are only tools in the arsenal for trusted software engineering. The proper deployment of the arsenal must be tailored to suit the needs of the particular application.

REFERENCES

[1] *ActiveX Consortium.* `http:/www.activex.org`.

[2] *Chrysalis, Inc.* `http:/www.chrysalis-its.com`.

[3] *IBM PCI Secure Co-processor.* `http:/www.ibm-.com/Security/cryptocards`.

[4] *Marimba Inc.* `http:/www.marimba.com`.

[5] *Spyrus Product Guide, Spyrus, Inc.* (See also: `http://www.spyrus.com`).

[6] Fips140-1 security requirements for cryptographic modules. Technical report, NIST, 1994. `http://csrc.ncsl.nist.gov/fips-/fips1401.htm`.

[7] R. Anderson and M. Kuhn. Tamper resistance – a cautionary note. In *Second Usenix Electronic Commerce Workshop.* USENIX Association, November 1996.

[8] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Noar. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994. Originally appeared in *FOCS* 91.

[9] Y. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. Referee: Trust management for

web applications. In *Proceedings of the Sixth International World-Wide Web Conference*, pages 227–238, 1997.

[10] P. Devanbu and S. Stubblebine. Automated software verification with trusted hardware. In *Twelfth International Conference on Automated Software Engineering*, November 1997.

[11] P. Devanbu and S. G. Stubblebine. Cryptographic verification of test coverage claims. In *Proceedings of The Fifth ACM/SIGSOFT Symposium on the foundations of software engineering*, Zurich, Switzerland, September 1997.

[12] P. Devanbu and S. G. Stubblebine. Stack and queue integrity on hostile platforms. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, California, May 1998.

[13] E. Felten. Princeton safe internet programming java/activex faq, 1997. `http://www.CS.Princeton.EDU/sip/java-vs-activex.html`.

[14] P. W. Fong. Modular verification of dynamically-loaded mobile code. *Working Paper*, August 97.

[15] J. Gosling, B. Joy, and G. Steele. *The Java^{TM} language specification*. Addison Wesley, Reading, Mass., USA, 1996.

[16] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine specification*. Addison Wesley, Reading, Mass., USA, 1996.

[17] G. McGraw and E. Felten. *Java Security: Hostile Applets, Holes & Antidotes*. John Wiley & Sons, 1997.

[18] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[19] G. Necula. Proof-carrying code. In *Proceedings of POPL 97*. ACM SIGPLAN, 1997.

[20] S. G. Stubblebine. Recent-secure authentication: Enforcing revocation in distributed systems. In *IEEE Computer Society Symposium on Security and Privacy*, Oakland, California, May 1995.

[21] *The Kimera Project*. `http://kimera.cs.washington.edu`.

[22] H.-P. V. Vliet. Mocha java bytecode decompiler, 1996. `http://web.inter.nl.net/users/H.P.van.Vliet/mocha.htm`.

[23] F. Yellin. The java native code api `http://java.sun.com/docs/jit_interface.html`, 1996.

Appendix—Terminology

Some terminology is presented here for convenience. We assume asymmetric (public-key) cryptography with public/private key pairs: *e.g.*, $K_{\mathcal{P}}^{-1}$ is a private key for the individual $\mathcal{P}$ and $K_{\mathcal{P}}$ is the corresponding public key.

Signatures Given a datum δ, $\sigma_{K_{\mathcal{P}}^{-1}}(\delta)$ is a value representing the signature of δ by $\mathcal{P}$, which can be verified using $K_{\mathcal{P}}$. Note that $\sigma_{K_{\mathcal{P}}^{-1}}(\delta)$ is usually just an encrypted hash value of δ. It is infeasible for $\mathcal{P}$ to find $\delta^{+} \neq \delta$ such that $\sigma_{K_{\mathcal{P}}^{-1}}(\delta^{+}) = \sigma_{K_{\mathcal{P}}^{-1}}(\delta)$. It is also infeasible to produce the signature $\sigma_{K_{\mathcal{P}}^{-1}}(\delta)$ from δ (verifiable against δ and $K_{\mathcal{P}}$) without knowledge of $K_{\mathcal{P}}^{-1}$. We advocate the use of such signatures by trusted agents to attest proven properties of software.

Certificates Given a public key K_{π} for an individual π, and a certifying agent ω with public key K_{ω}, the signature $\sigma_{K_{\omega}^{-1}}((K_{\pi}, \pi))$ is taken as a (feasibly) unforgeable assertion by ω that K_{π} is the public key of π. This is called a certificate, denoted here by $Cert_{\omega}(K_{\pi})$ and is used in security infrastructures as an introduction of π by ω to anyone who knows K_{ω}. A trusted *certificate authority* with well-known public key can be used as a repository of keys and a source of introductions. By composing certificates, chains of introductions are possible. A similar mechanism can be used for a *key revocation*, which is just a signed message from an authority indicating that a public key is no longer valid. We use certificates and revocations to introduce and revoke trusted software verifiers. In general, a certificate is a kind of *credential* from an authority. Thus, an authority ω may generate a credential signed with K_{ω}^{-1} for a trustworthy software tool τ of the form "K_{τ} is the private key for τ; I also believe that software signed by τ can be trusted to not delete files not in `/tmp`". With this credential, an agent $\mathcal{H}$ can verify software signed by τ and then perhaps allow the software to run, writes to `/tmp` are acceptable.

Key Management Given a set of certificate authorities, and a set of other participants, it is possible to set up a policy by which keys are introduced and revoked by certificates. We advocate use of such policies for configuration management.