

Research directions for automated software verification: Using trusted hardware

Prem Devanbu & Stuart Stubblebine
Information Systems and Services Research Center,
AT&T Labs— Research
Florham Park, NJ 07932, USA
prem,stubblebine@research.att.com

Abstract

Service providers hosting software on servers at the request of content providers need assurance that the hosted software has no undesirable properties. This problem applies to browsers which host applets, networked software which can host software agents, etc. The hosted software's properties are currently verified by testing and/or verification processes by the hosting computer. This increases cost, causes delay, and leads to difficulties in version control. By furnishing content providers with a physically secure computing device with an embedded certified private key, such properties can be verified and/or enforced by the secure computing device at the content provider's site; the secure device can verify such properties, statically whenever possible, and by inserting checks into the executable binary when necessary. The resulting binary is attested by a trusted signature, and can be hosted with confidence. This position paper is a preliminary report that outlines our scientific and engineering goals in this project.

1 Background

With the advent of programmable network applications such as the world wide web, trust in content hosting has become an important aspect of distributed computing. Providers (henceforth denoted \mathcal{P}) build content (C) (applets, servlets, switchlets, aglets, CGI bins, etc). Hosted content is a kind of “mobile code”. Hosting companies (\mathcal{H}) such as AT&T install this content on their networks. Clearly, wrong or malicious software could have a devastating impact on a hosting businesses' machines and network. Hosting organizations need assurance that the software has a certain set of properties, and/or that the vendor has used certain processes. These properties can be verified by ex-

tensive testing on \mathcal{H} 's part, which can give us some beliefs subject to confidence levels. The typical scenario is shown below.

1. \mathcal{P} sends C to \mathcal{H} .
2. \mathcal{H} tests C under many different possible conditions.
3. If C passes tests, \mathcal{H} agrees to host software.

Other options for \mathcal{H} include formal verification, transforming the program in some way (by inserting asserts that fail when safety properties are violated, e.g.), or sandboxing [14] the software in a runtime environment. All these options have hurdles, including difficulties in verification, unwanted information disclosure, a priori human costs, administrative difficulties, or runtime penalties.

2 Related Work

The issue of mobile code has received a great deal of attention lately [3]. In addition to the naive testing approach described in the previous section, there are approaches based on formal program verification. Verification has to be carried out by the hosting computer, since in general, any type of tool that runs on \mathcal{P} 's machine cannot be trusted. Even if such tools are provided by \mathcal{H} , there is no guarantee that such a tool will not be compromised in some way at \mathcal{P} 's site, and thus be made to produce bad output.

If \mathcal{H} has access to the source code for C, then more options become possible. First, typically, source has more information than the executable, so it becomes possible to check certain properties statically. Some properties cannot be statically checked, and must be done at run time; in such cases, access to the source code allows increased precision in determining exactly where run time checks need to be inserted. Unfortunately, \mathcal{P} may be unwilling to release source to \mathcal{H} . Of course, \mathcal{P} can run these static checks on the source

code at his machine, and make various claims, but H has no means of verifying such claims.

In this section, we discuss some approaches based on static typing, formal verification, and cryptographic signatures.

2.1 Java

Java supports “trusted” hosting of content. Java applets constructed by potentially hostile parties can be hosted in the carefully controlled Java runtime environment. Java security is based on a number of principles [7], one of which is type safety. Verifying type safety involves a great deal of information about a program; for this reason, Java byte codes contain exactly the same information as Java source code. This byte code verification is done every time an applet is received, before it can be executed. When an applet’s bytecodes have been successfully verified, there is a reasonable (though as yet formally unproven) belief that the bytecodes will not give rise to type errors during execution. Thus security flaws due to type confusion are avoided, and illegal access to facilities (e.g., calling a private method of a class from a method of another class) are guaranteed not to happen.

There are several difficulties associated with this approach. First, for large applets, there can be a significant overhead to doing byte code verification. Second, since the safety property checks (in this case, type checking) are done at the hosting party’s site, exploitable weaknesses (several are described in [7]) in the checking process must be addressed by propagating new versions of the verifier to every user of the web client. The large, diverse population of web users makes it unlikely that timely and orderly update of faulty versions of the verifier will occur. Finally, Java byte code *is* source code; bytecode decompiler tools such as mocha [13] have amply demonstrated this fact. This naturally raises intellectual property protection concerns: full implementation details are revealed to every user.

2.2 Proof-carrying Code

The traditional formal verification approach involves constructing (given a piece of software) a formal proof that certain safety properties hold. Typically such proofs are large, cumbersome, and make use of powerful proof techniques such as induction. There are formidable obstacles to the automated construction of such proofs. It would be unrealistic for \mathcal{H} to construct such proofs about every hosted software C .

Necula [8] advocates an appealing variant of the formal verification approach. The proof Π is constructed by \mathcal{P} , and is shipped to \mathcal{H} along with the binary program C . C is suitably annotated to facilitate the construction of a ver-

ification condition V , with the property that if V could be established, the safety property desired by \mathcal{H} holds. Π is constructed to establish V in a formal framework that \mathcal{P} and \mathcal{H} have agreed upon. The burden on \mathcal{H} is now reduced to the far easier task of *proof checking*, which can often be carried out very fast. Gunter *et al* [6] recommend an allied approach they call “proof referencing” code, which involves the development and reuse of logical frameworks constructed as HOL theories. Such frameworks reduce the proof construction burden on \mathcal{P} . However, it is important to bear in mind that the proof-carrying code approach is not always applicable. Not all safety properties can be cast as a verification condition [10, 12].

These approaches are subject to some of the same issues discussed in § 2.1. First proof checking can still be a substantial task, depending on the size of the proof; Necula reports that a type-safety proof for a 730 byte binary program was checked in 1.9 ms. For many practical programs, this overhead can be expected to be a good deal larger. In cases where the proof is checked once and the program is run many times, the overhead may be acceptable; in the case of highly mobile code such as applets and agents, this may represent an unacceptable overhead. Second, proof checkers are installed and run on the hosting computer, and thus are subject to the version control and release problems discussed earlier. Finally, the full proof, and any invariants that contribute towards the proof will have to be released to \mathcal{H} for checking; this may lead to unwanted loss of intellectual property.

2.3 “Trusted Builder Signatures” the ActiveX model

The ActiveX model [1] also allows for the possibility of automating the development of trust in software that is received over the internet. ActiveX is based on the assumption that software built by well-known individuals and companies can be trusted. Authenticity of software is established by an attached cryptographic signature. If the key on the signature corresponds to one of “trusted group” then the software is accepted and run on the machine without any further static or dynamic checks. The advantage of this approach is that there is no static verification or run time checking. Signature checking is very quick, and adds very little overhead.

The trust in this case is entirely based on proper key management, and proper use of the keys in signatures. If the keys get stolen or misused, signed (and therefore completely trusted) software could wreak havoc. To quote the Princeton Safe Internet Programming group [5]:

“ActiveX security relies entirely on human judgement. ActiveX programs come with digital signatures from the author of the program

and anybody else who chooses to endorse the program.”

The basic problem here is that the signatures in ActiveX convey an unspecified, unverified intention on the part of the signer. Why did the signer sign a particular piece of software? What does she mean by the signature? Who else has her signing key? The confusion over these critical issues lies at the heart of the ActiveX signature-based trust model. 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 to be limited from using code produced by an unknown software producer (even though the code may have been signed by this producer).

2.4 Third-Party Verifiers

Software producers can have their code tested by third-parties presumably trusted by the consumer. The third-parties cryptographically sign the code attesting to the tests the code has passed.

This approach, which may be attractive to some software vendors may not be suitable for everyone for a variety of limitations. The software producer must trust the tester to maintain confidentiality over aspects of the testing process. There are any number of aspects that can be a secrecy concern to the software producer including the fact that a product has been submitted for testing, the results of the testing, and information revealed about the code during testing. Other approaches, not involving third party testing [4], could be used. Testing approaches, however, have a fundamental limitation: it is impossible to test programs under all possible conditions. Thus it is always possible that a program may exhibit undesirable behaviour under conditions that were never raised during testing.

2.5 Synthesis

Our goal is to combine the efficiency and speed of signature verification with the semantic force of formal verification, while simultaneously providing for better key management, particularly relatively fool-proof mechanisms to prevent the theft and/or misuse of signing keys. We also seek to avoid disclosure (to any third parties) about the conduct of testing or verification processes. We accomplish this using widely known, established devices used currently in electronic commerce applications.

3 Physically Secure co-processors.

A physically secure co-processor is a device that is a computer that is encased in a tamper-proof enclosure. The enclosure is designed to enable the processor to self-destruct (erase all its memory) if tampered with. Such a processor can be made to contain a certified private key. Such processors are available in various configurations, ranging from smart cards with limited abilities to processors built on physically sealed circuit packs. We refer to such devices generically as \mathcal{PSC} (physically secure co-processor).

A \mathcal{PSC} comprises a CPU, ROM and RAM memories, possibly some specialized hardware for cryptography, an interface with limited (untrusting) access, all enclosed in a secure, tamper-proof enclosure. They are available currently with upto 1MB RAM, and a secure operating system, which can run different applications. The entire \mathcal{PSC} is readily portable, and can be installed in different machines (as long as the appropriate docking port is available). Software (certified with appropriate signatures) can be downloaded in the \mathcal{PSC} and executed. In this manner, a distributed computation, can be conducted, with some secure computations conducted in the \mathcal{PSC} and normal computations in the hosting machine. Independent agents can develop trust in these computations, subject to signatures appended to the results of the computations conducted by the \mathcal{PSC} . Details of physically secure co-processors, and many different applications (other than the one discussed herein) can be found in [15].

4 Tools Resident in Physically Secure co-processors.

Our approach is to establish trust in software using tools resident in a \mathcal{PSC} installed at the \mathcal{P} 's site. Specifically, we envision tools resident in \mathcal{PSC} devices. The \mathcal{PSC} would be delivered to \mathcal{P} 's site; the tool which checks the \mathcal{P} 's source code for the desired property would either reside on the \mathcal{PSC} or could be delivered later to \mathcal{P} with the signatures necessary to develop trust in the \mathcal{PSC} .

First, the tools process the source code and verify the desired property. Then using a key resident in the \mathcal{PSC} , they build the binary, append a statement to the effect that the binary can be trusted to possess the desired property and sign it with the \mathcal{PSC} 's key. Key certificates could also be appended. Upon receipt of this binary, with the accompanying signature, the hosting organization can trust that the binary has the requisite property. Both software producers and software hosting organizations gain significant advantages; details are discussed below.

4.1 Java Application

An immediate application of this technology is to the verification of Java byte codes. Java byte codes can be constructed by an untrusted developer, on an untrusted machine. Once the development is complete, the developer can “submit” the byte codes to a byte code verifier, *resident in a \mathcal{PSC}* , for verification. When the byte codes have been verified, the \mathcal{PSC} outputs a signature certifying that the byte codes have been verified. Because of the confidence in the physical security of the \mathcal{PSC} and its resident byte code verifier, browsers can simply check the signature and run the byte codes. Signing keys can be managed via certificates embedded within the browser, supplied by the \mathcal{PSC} , or pushed to the browser via “push” technologies etc. Similarly, key revocations can be pushed to the browser or pulled from a central repository at startup time. Pushing verification to the bytecode developer’s site, and only signature checking at run time offers a performance advantage over the existing approach of always checking bytecodes prior to execution. There is also another significant advantage.

Security weaknesses in the Java type system or the byte code verifier have been discovered on several occasions [7]. When such weaknesses are discovered, the only way currently for browsers to secure themselves from attacks that exploit a known weakness is to download and install a new version of the browser (or applet viewer). There are obvious logistical difficulties with $O(10^7)$ browser users updating their browsers; many browsers are unsophisticated, and may simply be unaware or unable to update their browsers, thus leaving themselves vulnerable. If the verification is done by the developer, the burden of upgrading to avoid known weaknesses now shifts to the developer; he has to get a new \mathcal{PSC} (or download certified new software into his \mathcal{PSC}). On the client side, upgrades, version management etc become transformed into a far more tractable key management issue. When a flaw is discovered in a version of the bytecode verifier, the keys corresponding to that version of the verifier can be revoked by the manufacturer; a certified revocation message can be passed on the browsers using “pull” or “push” approaches [11].

4.2 Formal Verification

Java bytecode verification is possible only because Java bytecodes are essentially source code. Approaches suggested by Necula [8] *et al* work on the binary; intellectual property is much harder to steal from binaries. However, in the proof-carrying code approach, the binary is accompanied by annotations and a proof. These are needed by the code consumer to generate both a safety verification condition, and to check that this condition is showed by the

proof. Unfortunately, the annotations and the proof can disclose a lot of valuable information about design and implementation. For example, the example given in [8] discloses the full layout of the datastructures used. In this particular case, there was only one datastructure; but in general, to establish that no bad pointers are generated by a given binary program, it would be necessary to disclose the layouts of all the heap datastructures and the invariants maintained in each loop of the program that pertained to memory usage. For example, suppose the loop contained a proprietary tree or graph traversal algorithm. The data layouts of the graph, and significant information about the operation of the algorithm would be disclosed via the loop and datastructure invariants. Commercial vendors may be reluctant to reveal this type of information.

Suppose a proof checker were embedded in a \mathcal{PSC} . Now a developer could generate a proof carrying binary, and reveal the annotations and the proof to a \mathcal{PSC} , at his site. Note that only a proof checker need to be embedded in the \mathcal{PSC} ; any tools actually used to construct the proof could be left on the \mathcal{P} ’s machine. Any associated theories or infrastructures that were used in the construction of the proof can be revealed to the \mathcal{PSC} for verification. Theories with derived theorems could be presented with proof for such theorems, or with a signature from a trusted verification authority.

The \mathcal{PSC} would generate the condition from the binary, check the proof, and sign the binary together a statement that the relevant safety property had been established. In this case, the proof and the annotations need not be revealed to the code consumer; the signature of the \mathcal{PSC} has the same import.

Of course the advantages discussed in the Java section, regarding version management and efficiency at the code consumer site would also apply here. Again, with binaries, there are significant performance and intellectual property protection gains.

4.3 Implementation issues

Important implementation considerations are the amount of computing resource available on the \mathcal{PSC} , the communication latency, and memory on the \mathcal{PSC} .

For the trusted tools application, the CPU speed probably not a major obstacle. Note that tools can be run on more powerful, untrusted machines first, to see if the results are satisfactory; only when the software vendor is ready to release the software is it necessary to actually run the source code through the \mathcal{PSC} . Since it has to be done only once for each release, slow CPU processing is probably not an issue. We now turn to the issue of memory and communication latency.

5 Leveraging the limited resources of the \mathcal{PSC}

\mathcal{PSC} s have somewhat limited amount of memory and high communication latency for accesses between the \mathcal{PSC} and the host. In order to conduct complex computations involving representations of programs and proofs, it becomes necessary to rely on the hosting computer which is potentially hostile. Any symbolic computation (such as proof checking or type code verification) typically involves the use of data structures such as stacks, queues, heaps, trees, hash tables, arrays, etc. Relying on the hosting computer to maintain data structures is acceptable in most applications provided we can reliably detect faults (whether malicious or otherwise).

We are exploring a number of schemes to detect violations of datastructure integrity. Invariants can be checked when performing operation on the data structure. Input to these invariants can be stored locally on the \mathcal{PSC} as well as on the remote host by checking cryptographic signatures on retrieved data. We have developed techniques for using the host to store and perform the usual operations on stacks, queues, hashes, arrays etc; with a very high probability, attempts by the host to corrupt these data structures can be detected. This is achieved by storing a constant or at most a logarithmic number of bits in the \mathcal{PSC} ¹. Our techniques do not aspire to the information-theoretic level of security of memory checking protocols described in [2]. In this application, the adversary (\mathcal{H}) enjoys just a constant level of speed up over the memory checker, and lower levels of security appear adequate. We are working a more precise formal characterization of the level of security achieved by our approach.

6 Conclusion

We have described an approach to program verification that combines the semantic import of type checking and formal program verification with the efficiencies of signature checking. The approach offers some attractive features, particularly in the area of protecting software developer's secrets, and for plugging holes in deployed software. The approach faces the obstacle of resource limitation in current \mathcal{PSC} s. We are exploring several approaches to leveraging the limited, but trusted resources in the \mathcal{PSC} to use the resources of the hosting computer in a trusted way. It is important to note that only code producers need to use a \mathcal{PSC} with the embedded trusted tools; code consumers merely check signatures based on keys (and any accompanying certificates) originating from the \mathcal{PSC} . Of course,

¹Logarithmic in the number of operations performed on the datastructure

if the hosting computer is compromised, then presumably the signature checking software may also be compromised, and using our approach offers no significant advantages or disadvantages.

Acknowledgements: We gratefully acknowledge Dave McAllester, Fred Douglass, Beto Avritzer and the anonymous reviewers for their suggestions and input.

References

- [1] *ActiveX Consortium*. <http://www.activex.org>.
- [2] 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*.
- [3] DARPA. Workshop on foundations of secure mobile code, March 1997. <http://www.cs.nps.navy.mil/-research/languages/wkshp.html>.
- [4] 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.
- [5] E. Felten. Princeton safe internet programming java/activex faq, 1997. <http://www.CS.Princeton.EDU/sip/java-vs-activex.html>.
- [6] C. Gunter, P. Homeier, and S. Nettles. Infrastructure for proof-referencing code. In *Proceedings, Workshop on Foundations of Secure Mobile Code*, March 1997. <http://www.cs.nps.navy.mil/-research/languages/wkshp.html>.
- [7] G. McGraw and E. Felten. *Java Security: Hostile Applets, Holes & Antidotes*. John Wiley & Sons, 1997.
- [8] G. Necula. Proof-carrying code. In *Proceedings of POPL 97*. ACM SIGPLAN, 1997.
- [9] R. Rivest, A. Shamir, and L. Adleman. A Method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 1978.
- [10] J. Rushby. Critical system properties: Survey and taxonomy. Technical Report CSL-93-01, Stanford Research Institute, 1993.
- [11] S. G. Stubblebine. Recent-secure authentication: Enforcing revocation in distributed systems. In *IEEE Computer Society Symposium on Security and Privacy*, Oakland, California, May 1995.
- [12] C.-R. Tsai, V. D. Gligor, and C. S. Chandrasekaran. On the identification of covert storage channels in secure systems. *IEEE Transactions on Software Engineering*, 1990.
- [13] H.-P. V. Vliet. Mocha java bytecode decompiler, 1996. <http://web.inter.nl.net/users/H.P.van.Vliet/-mocha.htm>.
- [14] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles*, 1993.
- [15] B. Yee and D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of The First USENIX Workshop on Electronic Commerce*, New York, New York, July 1995.

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 [9] 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 $K_{\mathcal{P}}^{-1}$. Such signatures on programs made by trusted verifiers in trusted hardware devices can be used in lieu of actual verification by each host.

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 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 keeping revoking the ability of faulty versions of verifiers to sign code, and to introduce new versions.