Proof-Carrying Code
Survivability Validation Framework

Andrew W. Appel
Princeton University

July 17, 2001

1 Technology Description & Security Problem Addressed

Our project applies automated proof checking to two application domains: protecting host computers from untrusted application programs, and distributed authentication for access-control.

Computers have many reasons to run untrusted application programs. Active-network routers may delegate decisions to embedded user code, e-mail recipients may want to run executable attachments, and high-assurance systems may want to run commercial off-the-shelf software for some applications. In all of these situations, the survivability goals are to protect private host data from modification or access by the untrusted code, to limit the code's access to external resources, and to prevent execution faults from crashing the host.

Two traditional and successful mechanisms that combine to achieve these goals are hardware virtual memory with an operating-system kernel. However, today's environment often requires the host to interact with the application through an object-oriented interface, which is very clumsy to do when address-space boundaries must be crossed: instead of pointers and private variables, we have remote-procedure call with marshalling/unmarshalling of data. Our *proof-carrying code* (PCC) and *certified binaries* approach will achieve security goals even in a shared-memory, rich-API environment. For the remainder of this report we will discuss proof-carrying code. The issues for certified binaries are similar, and we will discuss the differences where appropriate.

In the realm of distributed authentication, the goal is to permit clients to access servers in an environment where they have not previously been "introduced" to each other. Public key infrastructures (PKI) are examples of distributed authentication frameworks. Previous approaches suffer from the problem that when the policy language is expressive enough to be useful, the

decision procedure for granting access becomes too complicated to be trustworthy. Our *proof-carrying authorization* (PCA) solves this problem.

The two technologies can be used independently or together, as they address different aspects of the security problem: PCC provides low-level confidentiality and integrity of a shared-memory address space, PCA provides confidentiality, integrity, and authentication of higher-level objects such as files, database transactions, and web services.

2 Assumptions

A1	Instruction-set architecture executes correctly (manufacturer correctly implements specification, no memory bit-errors, no attacks by voltage variation, etc.)
A2	Capability management: host's access control policy, written by host administrator in our expressive policy language, is appropriate to host's needs

3 Threats, Attacks, Vulnerabilities

There are several ways that malicious code can threaten the security of a host system [DFW96].

Design

TAV-1.1	Exploitable inconsistency in policy
TAV-1.2	Erroneous decision procedure for granting access or running untrusted program

Implementation

TAV-2.1	Bug in implementation of protection mechanisms
TAV-2.2	Bug in implementation of decision procedure

Operation

TAV-3.1	Client code dereferences address outside its own space
TAV-3.2	Client code jumps to address outside itself that's not an API entry point (bypassing access controls)
TAV-3.3	Inconsistency in link-loading name resolution
TAV-3.4	Client code doesn't execute what's checked
TAV-3.5	Forging of certificates
TAV-3.6	Attacker uses compromised keys

4 Survivability and security goals

Proof-carrying code assures integrity (I) by protecting the host from unauthorized memory writes and unauthorized API calls; it assures confidentiality (C) by protecting against unauthorized memory reads and unauthorized API calls. Proof-carrying authentication provides a framework for authentication (AU) in a distributed system; proof-carrying authorization can be used to specify policies that protect confidentiality (C) and integrity (I). Our project has not addressed nonrepudiation (NR) or availability (AV).

In addition to the survivability and security goals, an important goal of our project is to provide flexibility to the designer of a component-based system. Proof-carrying code allows the untrusted component to interact with the host using a convenient, expressive, object-oriented interface. Proof-carrying authentication/authorization has a very expressive policy metalanguage which allows specification of application-specific policy languages and policies in a wide range of domains. We will denote these "flexibility" goals using the letter F.

5 Comparison with other systems

Traditional operating systems with virtual memory do well at protecting availability, confidentiality, and integrity from threats TAV 3.1-3.4, but are only moderately strong against TAV-2.1. Conventional operating systems do not provide much flexibility (F) in interfaces and policies.

Threats versus Goals for Operating Systems with Virtual Memory

		AV	I	C	AU	NR	F
Design	TAV-1.1		☹	☹			☹
	TAV-1.2						
Implementation	TAV-2.1		☹	☹			☹
	TAV-2.2		☹	☹			
Operation	TAV-3.1		✓	✓			
	TAV-3.2		✓	✓			
	TAV-3.3		✓	✓			
	TAV-3.4		✓	✓			
	TAV-3.5						
	TAV-3.6						

Flexibility is a design and implementation issue, but not a run-time issue, so there is no F column for these rows.

In the goal-threat matrix, a blank square indicates that the threat is irrelevant to the goal. The ✓ symbol indicates that one or more mechanisms successfully defend the goal against the threat, and the ☹ symbol means that the goal is not well defended.

The Java Virtual Machine is a software-based approach. Incoming application code is type-checked by a *byte-code verifier*. Since Java has a type-safety property, type-safe code cannot access (or jump to) arbitrary addresses. After type-checking, the byte-codes are translated into native machine code by a just-in-time compiler, then run in a shared address space with the host API. Java provides fairly good flexibility in interfaces (F), but has has proven weak in defending integrity against TAV-2.1, 3.3, 3.4.

		AV	I	C	AU	NR	F
Design	TAV-1.1						✓
	TAV-1.2		☹	☹			
Implementation	TAV-2.1		☹	☹			✓
	TAV-2.2		☹	☹			
Operation	TAV-3.1		✓	✓			
	TAV-3.2		✓	✓			
	TAV-3.3		☹	☹			
	TAV-3.4		☹	☹			
	TAV-3.5						
	TAV-3.6						

Threats versus Goals for Java Virtual Machine

Code signing, as in ActiveX or signed Java applets, depends on a trust relationship. Code signing supports flexible interfaces (F) quite well. Although in principle it could support flexible authentication policies, as implemented in practice the signature is all-or-nothing, so the policy must be one-size-fits-all. Code signing is weak against certificate-based or key-based attacks (TAV-3.5, 3.6). And it defends integrity fairly well against malicious attackers, but it does nothing to protect against ordinary programming bugs by application developers (TAV-3.1, 3.2).

		AV	I	C	AU	NR	F
Design	TAV-1.1						✓
	TAV-1.2						
Implementation	TAV-2.1		✓	✓			✓
	TAV-2.2						
Operation	TAV-3.1		☹	☹			
	TAV-3.2		☹	☹			
	TAV-3.3						
	TAV-3.4		✓	✓			
	TAV-3.5		☹	☹			
	TAV-3.6		☹	☹			

Threats versus Goals for Code Signing

As Section 7 will show in detail, proof-carrying code and proof-carrying authentication support integrity, confidentiality, authentication, and flexibility quite well.

Threats versus Goals for Proof-Carrying Code and Proof-Carrying Authentication

		AV	I	C	AU	NR	F
Design	TAV-1.1		✓	✓	✓		✓
	TAV-1.2		✓	✓	✓		
Implementation	TAV-2.1		✓	✓	✓		✓
	TAV-2.2		✓	✓	✓		
Operation	TAV-3.1		✓	✓			
	TAV-3.2		✓	✓			
	TAV-3.3		✓	✓			
	TAV-3.4		✓	✓			
	TAV-3.5				✓		
	TAV-3.6				✓		

The threats/goals matrix for certified binaries is similar to the one shown here for proof-carrying code, but the quantitative analysis would be somewhat different, as we will describe.

6 Mechanisms and Assumptions

M1	Prover: constructs safety proof for untrusted application binary [Nec97]
M2	Machine specification: axiomatizes behavior of machine instructions [MA00]
M3	Safety policy: defines “theorem” to be proved [App01]
M4	Proof checker: determines whether proof matches theorem [PS99]
M5	Policy modeller: validation technique for safety policies [AF01]
M6	Semantics of types: safety proofs for advanced type systems [AF00]
M7	Digital signatures: can be generated only by holder of private key
M8	Expiration: “freshness dating” of certificates helps limit damage from key leakage

7 Rationale

		AV	I	C	AU	NR	F
Design	TAV-1.1		A2,M5[1]				M1,M3,M6[2]
	TAV-1.2		M4[3]				
Implementation	TAV-2.1		TCB[4]				M1,M3,M6[2]
	TAV-2.2		M4[3]		M4[5]		
Operation	TAV-3.1		M2,M3,M4[6]				
	TAV-3.2						
	TAV-3.3		note[7]				
	TAV-3.4		M2,M3,M4[8]				
	TAV-3.5				M7		
	TAV-3.6				M8		

Footnotes for Threat matrix:

0. All defenses below rely on assumption A1 (correct execution of machine instructions), which is not listed separately for each.

1. Defense against TAV-1.1 (inconsistency in policy) by M5 (policy modeller). Appel and Felten [AF01] have described how to construct machine-checkable proofs of the consistency of a wide variety of safety and security policies. Still, policy modelling requires intelligent use by the policy designer, hence assumption A2.

2. Expressive, efficient, object-oriented interfaces between host and application are achieved by M1,M3,M6. A shared-memory interface between client and host can be very efficient (no hardware context switch) and expressive (objects with methods, instead of marshalling/unmarshalling data for remote procedure call). Such interfaces are secure only if private variables and private methods are respected, as addressed by goals G1 and G2. Our mechanism for semantic modelling of types (M6) allows the safety policy M3 to require, the the prover M1 to provide, proofs that the application respects its APIs. Our policy language for APIs [AF00] and for access control [AF01] is expressive and flexible.

3. Defense against TAV-1.2 (bugs in decision procedure). Unlike the Java Virtual Machine, where the decision procedure (byte-code verifier) is quite complex, we have moved almost all of the complexity out of the checker (M4) and into the prover (M1). Our decision procedure (M4) is approximately 1000 lines of code, its rationale is well understood and described in the scientific literature [HHP93], and there are multiple independent implementations available.

4. Defense against TAV-2.1 (bug in implementation of protection mechanisms) is by the very small trusted computing base. That is, the implementation of M2,M3,M4 totals less than 4,000 lines of code. In comparison, other systems (operating-system kernel, Java just-in-time compiler) have mechanisms that are 40,000 to 1,000,000 lines of code. We believe it is possible to construct 4,000 lines of code free of significant bugs, but making 40,000 or a million lines bug-free is practically impossible.

   The size of M1 (prover) is much larger, perhaps over 100,000 lines of code. However, bugs in M1 cannot compromise any of the security goals. M1 "fails safe," in the sense that ill-constructed proofs will be rejected by the checker (M4).

5. Unlike X.509 [Int97] and SPKI [EFL+98], our distributed authentication protocol has a cleanly specified decision procedure based on the fundamentals of logic.

6. Confidentiality and integrity are defended against threats TAV-3.1, TAV-3.2 (dereferencing unauthorized addresses, jumping to unauthorized entry points) by M2,M3,M4 (machine specification, safety policy, proof checker). That is, the safety policy defines the sets of authorized addresses for reading, writing, and jumping; the machine axioms for load, store, and jump refer explicitly to these sets; and the proof checker makes sure that the safety proof matches both the application program and the safety policy.

   In constructing the safety proof, mechanism M1 (prover) is useful but not required; the client may choose to use an alternate mechanism to construct a proof. For example, if the source code is written in a type-safe language, then an automatic prover such as M1 is appropriate, but for (small) assembly-language programs, hand-constructed proofs may be useful instead.

7. Protection against inconsistency in link-loading name resolution may be addressed using Necula's strategy of safety-checking the program after linking and loading, but at this early stage of implementation we have not yet decided this issue.

8. Defense against TAV-3.4 (client code doesn't execute what's checked) by M2,M3,M4, in the following sense: Unlike a Java-based system, where the byte codes are checked but the native machine code (produced by JIT) is what executes, proof-carrying code checks the same machine-code that executes.

7.1 Certified Binaries

The analysis for certified binaries is similar to the one for proof-carrying code. However, the size of the checker (M4) for certified binaries is significantly larger (perhaps by a factor of 10). Therefore,

it will be somewhat weaker against threats TAV-1.1 and TAV-2.1, though still significantly better against these threats than competing technologies such as operating systems and Java Virtual Machines.

8 Residual risks, limitations, caveats

Our "foundational" approach to proof-carrying code may not scale to full-featured programming languages. Certified Binaries will scale more easily.

Prover or checker may be too slow in practice, though good results have been obtained by others in a different version of proof-carrying code [CLN+00].

Compiler/prover technology may be too complex for development outside academia.

Covert channels can leak information; we analyze direct memory access, but not generalized information flow.

Measurements of Trusted Computing Base (i.e., the 4,000 lines discussed in note 4, page 7) do not include the C compiler that compiles the checker, nor the C compiler that compiles that compiler, ad infinitum [Tho84].

9 Costs

Previous sections have compared the benefits of proof-carrying code (and proof-carrying authentication) to competing technologies. Here we will discuss the costs of PCC relative to its competitors.

Speed. Operating systems with virtual memory give very good performance for ordinary program execution, but have significant cost in crossing the interface between application and host. Standard JVM (Java Virtual Machine) technology has good interface-crossing performance, but (typically) a factor of two or more slowdown in ordinary execution, even with a just-in-time compiler. Signed code can have excellent performance both on ordinary execution and at interfaces. Proof-carrying code has excellent interface-crossing performance and good-to-excellent ordinary execution performance.

Software development. Operating systems with virtual memory can use COTS (commercial off-the-shelf) components. The JVM approach requires that application development be in the Java programming language; normally this will not be more expensive than development in

other languages, and will often be cheaper, but it does prevent the use of legacy applications. Code signing requires careful auditing before signing, especially because there is no "just-in-case" automatic barrier as there is with the other technologies. The proof-carrying-code approach imposes constraints similar to the JVM approach: application development must be in Java or another type-safe source language, but then the prover/checker is transparent to the application developer.

Technology development. Operating systems are a known and developed technology. To augment an operating system so as to enforce a custom access-control policy is not difficult. JVM technology is not as mature, but there are some commercial systems that might be usable as infrastructure (subject to the security risks described in Section 5). Proof-carrying code is a newer technology; research results are very promising but further development will be required before use in real applications.

Certified Binaries technology may be somewhat easier to reduce to practice than proof-carrying code, so its development costs may be lower.

References

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253. ACM Press, January 2000.

[AF01] Andrew W. Appel and Edward W. Felten. Models for security policies in proof-carrying code. Technical Report TR-636-01, Princeton University, March 2001.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science (LICS '01)*, page (to appear). IEEE, 2001.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*. ACM Press, June 2000.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of 1996 IEEE Symposium on Security and Privacy*, May 1996.

[EFL+98] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. Simple public key certificate. Internet Draft draft-ietf-spki-cert-structure-05.txt, 1998.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[Int97] International Telecommunications Union. ITU-T recommendation X.509: The directory: Authentication framework. Technical Report X.509, ITU, www.itu.int, 1997.

[MA00] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, June 2000.

[Nec97] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999.

[Tho84] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.