JVM TCB: Measurements of the Trusted Computing Base of Java Virtual Machines

Andrew W. Appel Daniel C. Wang

Princeton University
April 12, 2002

Abstract. The trusted computing base (TCB) of a Java virtual machine (JVM) is the part of the program code in which programming bugs could lead to security holes. Java systems keep the front end compiler (which translates source code to byte code) out of the TCB, by having the JVM verify the safety of the byte code before just-in-time (JIT) compiling it to machine code. Still, the JIT compiler itself is usually in the TCB, and the more lines of code in the TCB, the more likelihood of security problems. We have measured the TCB size of several JVMs, and find that they range from 36,000 to 229,000 lines of source code.

When you obtain a piece of software – a shrink-wrapped application, a browser plugin, an applet, an OS kernel extension – you might like to ascertain that it's safe to execute: for example, it accesses only its own memory and respects the private variables of the API to which it's linked. The Java byte-code verifier can make such a guarantee by "type-checking" the byte-code program before the just-in-time (JIT) compiler translates it to machine code for efficient execution.

A safe mobile code host is one that permits untrusted software to run while guaranteeing that it obeys a given security policy; a JVM is an example of a safe host. The design of Java (or of any type-safe language such as C#, ML, Modula-3) allows this guarantee to be neatly decomposed into two parts:

Type safety. The Java byte-code verifier checks type-safety: that the program respects its API's, doesn't cast integers to pointers, doesn't fetch private fields, and so on.

Capability engineering. Type-safe applets interact with the outside world (and sensitive data) by politely making API calls (not impolitely modifying the private variables of API implementations). But the API must grant capabilities (implemented as objects) only in accordance with the security policy [Wal99, AWF00].

But suppose there's a bug in the implementation of the JVM? Then an attacker can design just the right kind of program that will exploit the bug and breach security. A bug in the verifier may permit type-unsafe code that casts integers to pointers. A bug in the JIT compiler may incorrectly compile type-safe byte-code into type-unsafe machine code in a way that the attacker can exploit to cast integers to pointers; and the same for bugs in the garbage collector. Once a type-safety violation is permitted, it's easy for the attacker to modify any memory address, and security is completely compromised [DFW96].

If there are no bugs that compromise type safety, there may still be bugs in the capability engineering. Then the attacker can't access and modify arbitrary memory locations, but might still be able to perform file and network I/O (by politely abusing API calls) that violate the security policy.

In assessing the vulnerability of systems to attack, we use the notion of *trusted computing base* (TCB), which is that portion of a system in which any bug might lead to a security hole. A bug outside the TCB may cause incorrect behavior but not insecure behavior. The untrusted software is outside the TCB; it may contain bugs that make it display meaningless or misleading results, but it must still obey the security policy (e.g., it cannot read or destroy the host's private data).

Many components of a JVM are not in the TCB. For example, user libraries, provided by the host to the untrusted code, implemented in type-safe Java, and that do not provide capability objects, cannot cause security holes even if they contain bugs. At worst they cause the untrusted code to behave incorrectly. This is an important advantage of type-safe languages; even a very large system can have a relatively small TCB.

A "high-assurance" safe host is one whose TCB has been carefully audited by appropriate methods such as mathematical proof, software engineering practices, or testing. Obviously, it is desirable that the TCB be as small and well-engineered as possible.

Just as the security guarantee decomposes nicely into two parts (in a type-safe language), the TCB can be separated into two parts. The *Safety TCB* includes any component in which a bug could compromise type safety: typ-

ically, the byte-code verifier, the just-in-time (JIT) compiler, the garbage collector, the core runtime system, and any libraries implemented in type-unsafe languages such as C or C++. In a conventional Java system, the type-safety audit is difficult, because the JIT is large and reasoning about its correctness is complex.

The *capability TCB* covers those APIs that allow access to resources such as files and networks. Fortunately, since most of these APIs are implemented in type-safe Java, the audit can take the source code at face value; this would not be possible in a C or C++ system, where there would be no protection from illegitimate casting by the untrusted applet.

Proof-carrying code (PCC) is a new approach to guaranteeing type safety, by constructing and verifying a mathematical proof about a machine-language program [Nec97]. With PCC, the byte-code verifier and the JIT compiler can be outside of the TCB, because the output of the JIT is verified. The TCB of a proof-carrying code system includes a verification-condition generator, logical axioms, typing rules, and a proof-checker. Fortunately, these TCB components are significantly smaller than the TCB of a JIT and JVM, and may be easier to reason about.

Foundational proof-carrying code is an improvement of PCC, being designed at Princeton, that is an attempt to answer the question, "What is the smallest possible TCB for a safe mobile-code system?" We reduce the size of the trusted components even at the expense of a large increase in the size of the untrusted parts [AF00, MA00, AM00, WA01].

Measurements. We can learn something about the design of a system by measuring the size (in lines of source code) of the TCB. This is a very crude measure, as it tells us little about the quality of the code, but it may nonetheless be instructive. We have measured several JVMs, and for each we report three numbers:

- *Safety TCB* measures a core Java system without its class libraries, roughly what is described in *The Java Programming Language* [AG96] minus chapters 11 and 12. This is meant to measure the parts of the system that ensure that Java classes respect the API's of the interfaces and classes to which they are linked – that is, type safety.
- *Security TCB* is the size of the entire system's trusted computing base; it approximates the safety TCB plus the capability TCB.

If we imagine a Java system with almost all the API's stripped away, leaving only the Object, Exception, and JNI classes, we get a "min-API" system. Any component not in the min-API system cannot be part of the Safety TCB; it will be either in the Security TCB or not in the trusted computing base at all.

For a conventional JVM with just-in-time (JIT) compiler, we calculate the number as follows. We count each module in one of the following categories:

FC Front-end compiler from Java byte codes to verifiable intermediate language; for conventional JVM's, where the verifiable IL *is* byte code, this is zero.

C Just-in-time compiler, byte-code verifier, linker.

GC Garbage collector.

CR Core run-time system (present even in the min-API system, necessarily present for any Java code to run, typically not implemented in Java itself).

UL Type-unsafe libraries to support APIs (not implemented in type-safe Java, not part of min-API system).

TL Type-safe security-critical runtime libraries (implemented in Java, provide sensitive capabilities such as I/O or security primitives, not present in min-API system).

L Type-safe security-irrelevant user libraries (implemented in Java, not called by security-critical libraries, not in min-API).

Then we can calculate

Safety TCB = C + GC + CR

Security TCB = C + GC + CR + UL + TL

For a JVM built using proof-carrying code, the same calculations apply; the difference is that the boundary between FC and C is moved; instead of being at the byte-code level, it is at the machine-code level. The C component now consists of the proof checker (and related components); C is much smaller, and FC is bigger.

We have measured the TCB size of several real systems and one hypothetical:

- *JDK* is Sun's Java Software Development Kit (Java 2 SDK 1.3)with its "classic" nonoptimizing just-in-time compiler (www.sun.com/software/java2/download.html).
- *Kaffe* is an open-source cross-platform JVM with a nonoptimizing just-in-time compiler (www.kaffe.org).
- *Hotspot* is Sun's Java 2 SDK 1.3 with with the "Hotspot" optimizing just-in-time compiler and a more efficient garbage collector (www.sun.com/software/communitysource/hotspot/download.html).

	JDK	Kaffe	Hotspot	Bullet Train	Ginseng	FPCC
FC [1]					42,000 [5]	200,000 [9]
C	38,200	38,200 [2]	204,500	116,200 [4]	25,000 [6]	2,650 [10]
GC	7,100	2,400	15,700	11,000	[7]	150 [11]
CR	8,900	7,200	8,900	7,600	6,000 [8]	800 [12]
UL	14,000	10,900 [3]	14,000	48,700	[8]	[13]
TL+L	14,500 [0]	16,600 [3]	14,500 [0]	13,100	[8]	[13]
L [14]		[3]			[8]	[13]
Safety TCB:	**54,200**	**47,800**	**229,100**	**123,800**	**34,600**	**3,600**
Security TCB:	87,200	[3]	257,600	185,600	[8]	[13]

[0] We do not include here 37,700 lines of embedded Javadoc comments.

[1] We do not include here the Java source-to-bytecode compiler, which is not part of the TCB and which usually operates at the code provider's site, not inside the JVM. The JDK's source-to-bytecode compiler is 31,000 lines of code.

[2] This number does not include the bytecode verifier, which is not yet implemented in Kaffe. To compute TCB totals, we estimate this size at 5,000 lines, which is the size of Sun's verifier [Mic01].

[3] Although most of the standard API is implemented, Kaffe's security manager is not. Therefore the size of the total Security TCB cannot yet be measured.

[4] This total includes 5,000 lines of libraries shared between the C and TL components; to avoid double counting, the number shown for TL does not include them. All numbers for BulletTrain provided by Kenneth Zadeck[Zad01].

[5] "Special-J" compiler, 33,000 lines; prover, 9,000 lines [CLN+00].

[6] Verification-condition generator, 23,000 lines; logical axioms, 700 lines; LF proof-checker, 1,400 lines [CLN+00].

[7] No number available. For computation of TCB sizes, we estimate at 3,600 lines (by analogy with Kaffe).

[8] Current size of core runtime [Ple01]. Libraries are incomplete, so Security TCB cannot be measured.

[9] This is a very rough estimate of the size of compiler, prover, and the non-TCB components of the garbage collector (see [11]).

[10] These are preliminary measurements of our Sparc opcode specification (600 lines) and instruction semantics (600 lines) [MA00], and safety policy (50 lines); plus the same kind of 1,400-line LF proof-checker used by Ginseng [CLN+00].

[11] Wang's *type-preserving garbage collector* [WA01] will be able to automatically verify almost the entire collector, leaving only 150 lines of support routines (interface to `memmap`) in the TCB.

[12] Our ability to verify machine code directly should allow most components of the core runtime to be moved out of the TCB, but still this number is more speculative than our other numbers.

[13] We have no implementation of APIs. Using the BulletTrain implementation as a model, a Security TCB of 65,000 lines seems plausible. Further research might significantly reduce the size of the Capability TCB.

[14] In this draft we have not yet separated L counts from TL.

Figure 1: Comparison of JVM TCB sizes

- *BulletTrain* is a conventional well-engineered high-performance JVM with optimizing JIT compiler, from NaturalBridge.com. We would expect it to achieve much faster performance than Kaffe, and probably faster than Hotspot.

- *SpecialJ* is a proof-carrying code JVM [CLN+00] developed by CedillaSystems.com. It probably achieves performance midway between Kaffe and BulletTrain, though in principle it should be possible to combine the compiler engineering of BulletTrain with the proof-carrying of SpecialJ.

- *FPCC* is a Foundational Proof-Carrying Code system being designed at Princeton University. It is still in design and construction, and the numbers we report here are a design target, not the measurement of an artifact.

Figure 1 shows the results. The size of the Safety TCB ranges from 139,000 lines (with a conventional optimizing JIT) down to 4,000 lines (with Foundational Proof-Carrying Code).

Conclusion. A conventional Java Virtual Machine with a nonoptimizing just-in-time compiler has a Safety TCB

of about 67,000 lines of source code, and total TCB of probably 130,000 lines.

A conventional, well-engineered JVM with optimizing compiler has a Safety TCB of about 140,000 lines of source code, and a capability TCB of 60,000 lines. A JVM using proof-carrying code technology has achieved a Safety TCB of 35,000 lines; this technology does not, at present, reduce the size of the capability TCB. The proposed "foundational proof-carrying code" technology might reduce the size of the Safety TCB to 4,000 lines.

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.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, Reading, MA, 1996.

[AM00] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. October 2000.

[AWF00] Andrew W. Appel, Dan Wallach, and Edward W. Felten. Safkasi: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, to appear, 2000.

[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.

[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.

[Mic01] Sun Microsystems. Java 2 SDK Standard Edition v1.3.0 for Solaris/Intel. //www.sun.com/software/communitysource/java2, January 2001.

[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.

[Ple01] Mark Plesko. personal communication, plesko@CedillaSys.com, January 2001.

[WA01] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 166–178. ACM Press, January 2001.

[Wal99] Dan Seth Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, Princeton, NJ, January 1999.

[Zad01] Kenneth Zadeck. personal communication, zadeck@NaturalBridge.com, January 2001.