Safe Kernel Extensions Without Run-Time Checking

George C. Necula and Peter Lee
Presented by Ian Murdock
October 11, 1999

1 Introduction

This paper describes a mechanism for allowing an operating system kernel to determine with certainty that it is safe to execute code supplied by an untrusted application program. The kernel first defines a safety policy and publishes it. An application that wishes to execute code in the kernel then submits its extension in a special form, called proof-carrying code, that contains a formal proof that the safety policy is obeyed. The kernel may then validate the proof, and if the validation succeeds, it knows the extension is guaranteed to respect the safety policy and may be safely executed in its address space.

Proof-carrying code is attractive because it provides an absolute guarantee that an extension may be executed safely, without the need for run-time checks, and without restricting the language in which the extension is written or the environment in which it is executed (beyond, of course, those restrictions needed to guarantee some specific notion of safety). Furthermore, proof-carrying code is tamperproof; tampering goes undetected only if the resulting code is still guaranteed to respect the safety policy (i.e., the code may still be executed safely).

2 Motivation

Applications often want to extend or specialize the functionality of the underlying operating system, either for performance reasons, or to meet certain functional needs. For example, based on knowledge of future access patterns, an application might want to influence how its data is cached, or how its blocks are laid out on disk, so as to improve the performance of future reads. Unfortunately, support for application-driven extensibility and specialization in existing general-purpose operating systems is limited at best and nonexistent at worst.

3 Approaches

Extensible operating systems are operating systems that may be changed to meet the needs of application programs. Typically, an extensible operating system allows an application to install code that interacts with the operating system kernel, augmenting existing system services and possibly implementing new ones. Because applications are allowed to install arbitrary code, a virtually limitless range of extensions may be implemented. However, because extension code is potentially unsafe, the operating system must guarantee that an extension cannot adversely affect the surrounding system in the event it misbehaves.

Extensible operating systems may be implemented in a number of ways. One approach is to place each extension in its own address space and rely on the virtual memory hardware for protection. This hardware-based protection comes at a high price, however, when extensions frequently interact with the kernel or each other, as each interaction potentially involves high-overhead protection boundary crossings and context switches. This approach is used by microkernels such as Mach.

Another approach is to interpret extensions in a limited environment, such a virtual machine, that only allows extensions to interact with the kernel in limited ways. Unfortunately, interpretation tends to be much slower than
executing code natively, and limitations in the extension language or environment may make it difficult to implement certain extensions. This approach is used by, among other things, the BSD packet filter system and Java.

A third approach is to implement extensions using a type-safe programming language, such as Java or Modula-3. An extension written in a type-safe language is inherently safe (as long as it is compiled properly) and may be safely executed in the kernel’s address space. The biggest limitation of this approach is the fact that it is language-specific; ideally, an extension could be written in any language. This approach is used by the Spin operating system, which allows extensions written in Modula-3 to be dynamically loaded into the kernel by application programs.

Finally, software fault isolation may be used to instrument compiled extension code so that it may be safely executed in the kernel’s address space. Software fault isolation is a simple and efficient technique, and because it operates on compiled code, it is independent of the implementation language. One disadvantage of software fault isolation is that it relies on run-time checks, which necessarily adds a small amount of overhead. Another disadvantage is that it may only be used to guarantee memory safety.

4 Proof-carrying code

Proof-carrying code is an alternative approach that does not depend on run-time checks or impose limitations on the extension’s language or execution environment beyond those necessary to enforce a certain notion of safety. Furthermore, proof-carrying code may be used to enforce arbitrary notions of safety, not just memory safety.

The operating system kernel first defines a safety policy and publishes it. An application that wishes to extend the operating system submits its extension to the kernel in a special form that contains a formal proof that the safety policy is obeyed. The kernel then validates the safety proof, after which it knows that the extension is guaranteed to respect the safety policy and may be safely executed in its address space. If the safety proof cannot be validated, the extension is unsafe and is rejected.

4.1 Safety policy

First of all, the operating system must have a formal way of expressing what it means for code to be “safe.” The kernel’s definition of safety is embodied in its safety policy.

4.1.1 Abstract machine

The first part of the safety policy is the abstract machine, a formal specification of the machine on which the extension will be executed, extended to support the notion of safety. An abstract machine is defined as a state-transition function that maps a machine state into a new machine state by executing an instruction. In addition to providing the usual semantics for each instruction, the state-transition function includes safety checks, predicates that must hold true in order for a transition to be valid. For example, to provide memory safety, the load instruction might check that it is safe to read from the memory word. Similarly, the store instruction might check that it is safe to write to the memory word. Thus, the abstract machine supports the safe execution of extensions, as defined by the safety checks.

4.1.2 Precondition

The second part of the safety policy is the precondition, a predicate in first-order logic that the kernel guarantees to be valid when the extension is invoked. Conceptually, the precondition may be thought of as a calling convention. In the case of memory safety, a precondition would specify which memory locations may be safely read and which may be safely written. The precondition might also specify, for example, that the extension must acquire a mutex before executing.

To illustrate, consider a kernel that maintains an internal table in which each process owns an entry that is comprised of two consecutive memory words: A tag word, and a data word. The tag word indicates whether or not the data word is writable, and the data word contains some information that a process might want to access or modify (e.g., a capability).
Assume that the kernel provides an interface for user processes to load extensions into the kernel and calls them with the address of their table entry in register $r_0$. In order for the extension to be executed safely, the following must hold true: (1) the extension may only access its own entry in the table and not the entries of other processes; (2) the tag word is read-only; and (3) the data word is read-only if the tag is zero. This safety policy may be expressed formally by the following precondition:

$$\text{Pre}_r = r_0 \mod 2^{64} = r_0 \land rd(r_0) \land rd(r_0 \oplus 8) \land \text{sel}(r_0, r_0) \neq 0 \Rightarrow wr(r_0 \oplus 8)$$

In other words, the contents of register $r_0$ is a valid register value\(^1\), it is safe to read the tag word at address $r_0$, it is safe to read the data word at address $r_0 \oplus 8$, and it is safe to write the data word at address $r_0 \oplus 8$ if and only if the tag is not 0.

### 4.1.3 Postcondition

Finally, the security policy may also specify a postcondition, a predicate in first-order logic that places additional conditions on the state of the machine after termination. For example, the postcondition might require that the extension release a mutex before terminating.

### 4.2 Certification

To prove that it is safe to execute an extension in the kernel’s address space, the application must prove that it may execute on the abstract machine without violating any of the safety checks, and that the postcondition holds true after the extension terminates. This process is known as certification.

An extension is certified by computing its safety predicate, a predicate in first-order logic that constitutes a formal statement that the extension will not violate any of the safety checks defined by the abstract machine. Conceptually, the safety predicate guarantees that, for any initial state that satisfies the precondition, the code may be executed safely (and, if it terminates, that the final state satisfies the postcondition, if one is specified). The safety predicate is then proven, and the proof is attached to the extension in a checkable form.

### 4.3 Validation

When an extension is loaded, the kernel extracts the code from the extension and computes its safety predicate in the same manner as during the certification step. Then, it checks that the safety proof included in the extension is a valid proof of the safety predicate. If it is, the kernel knows with certainty that the code in the extension may be executed safely. If it is not, the code is deemed unsafe and is rejected.

An important feature of proof-carrying code is that it is tamperproof. If the code is modified, then its safety predicate changes, invalidating the safety proof and causing the extension to be rejected. If the proof is modified, then it will either be invalid, or it will not correspond to the safety predicate computed by examining the code, also causing the extension to be rejected. If the code is modified in such a way that the safety predicate is unchanged, the code may still be executed safely.

### 5 Conclusion

Much work remains to be done before proof-carrying code can be applied on a large scale. The primary difficulty is the creation of the safety proofs, which is, in general, undecidable. In addition, the presence of loops often requires human intervention during proof creation; in other words, the creation of all but the most trivial of proofs is not yet a fully-automated process. Finally, the size of proofs are a concern; in general, they may be exponentially large. More experience is needed to know if this will be a problem in practice.

---

\(^1\)The Alpha is a 64-bit processor, so a valid register value is between 0 and $2^{64} - 1$, with negative values represented using two’s-complement representation. This constraint may be expressed formally by the equation $r_i \mod 2^{64} = r_i$. 

3
Nevertheless, proof-carrying code is an interesting technique that holds some promise as a means of building extensible operating systems. It allows extensions from untrusted applications to be loaded into the kernel’s address space and executed with little or no overhead beyond the one-time validation of an enclosed proof. The initial experiments are promising, but more experience with it is needed to know if it is a practical approach to building real systems.