Malicious Code Detection for Open Firmware

Frank Adelstein, Matt Stillerman
ATC-NY
33 Thornwood Drive, Suite 500
Ithaca, NY 14850-1250, USA
{fadelstein,matt}@atc-nycorp.com

Dexter Kozen
Department of Computer Science
Cornell University
Ithaca, New York 14853-7501, USA
kozen@cs.cornell.edu

Abstract

Malicious boot firmware is a largely unrecognized but significant security risk to our global information infrastructure. Since boot firmware executes before the operating system is loaded, it can easily circumvent any operating system-based security mechanism. Boot firmware programs are typically written by third-party device manufacturers and may come from various suppliers of unknown origin. In this paper we describe an approach to this problem based on load-time verification of onboard device drivers against a standard security policy designed to limit access to system resources. We also describe our ongoing effort to construct a prototype of this technique for Open Firmware boot platforms.

1. Introduction

Our critical infrastructure for transportation, communication, financial markets, energy distribution, and health care is dangerously dependent on a computing base vulnerable to many forms of malicious attack and software failure. The consequences of a coordinated attack on our information infrastructure could be devastating [22]. One serious vulnerability that has largely been ignored up until now is *malicious boot firmware*.

Most computing devices are powered up by a *boot sequence*—a series of computational steps in which the hardware is initialized and the operating system loaded and started. *Boot firmware* is the program that controls this process. Boot firmware typically runs in privileged mode on bare hardware. It has essentially limitless access to peripheral devices. The boot program runs before the operating system is loaded, prior to the start of most security measures. Thus malicious boot firmware has the potential to cause very serious harm. This harm falls into three general categories:

- It could prevent the computer from booting, thus effecting a denial of service.
- It could operate devices maliciously, thereby damaging them or causing other harm.
- It could corrupt the operating system as it is loaded.

This last form of attack is perhaps the most serious, since most other security measures depend on operating system integrity. Even the most carefully crafted security mechanisms implemented at the operating system, protocol, application, or enterprise levels can be circumvented in this manner.

On a typical computing platform, the boot firmware is composed of many interacting modules. There is usually a *boot kernel*, which governs the bootup process, as well as boot-time device drivers supplied by the manufacturers of various components. The purpose of a boot driver is to initialize the device, perform diagnostic checks, establish communication with other devices connected to it, allocate system resources, and other similar tasks. The driver often resides in ROM on the device itself and is loaded and run at boot time.

To interact successfully, these pieces must respect well-defined abstraction boundaries and communicate only via standardized interfaces. Yet at boot time, the pieces all run in the same address space in privileged mode. There is no isolation and no external enforcement of good citizenship. It would be well within the means of determined opponent to introduce malicious code into a device driver for a keyboard or mouse, for example.

One line of defense is to ensure the integrity of firmware via digital signatures [2] or chain-of-custody and physical protection. This strategy requires that we assume that the boot firmware was originally benign. Such a belief could be based on trust in the supplier or in some detailed examination of the code. It simply ensures that the code has not been changed after it was approved. Thus, the strategy is a means for preserving an existing relationship of trust, but not of establishing trust.

This strategy could be costly in practice. There may be a large, far-flung network of vendors for whom trust must be established. Moreover, there are mechanisms for automatically updating device drivers and firmware with patches via the Internet. Firmware that is updated regularly would need to be reexamined each time.

In this paper we describe an alternative technique that provides a basis for trust in boot firmware, regardless of its source. The technique involves automatic verification of boot firmware modules as they are loaded. We also describe ongoing work to construct a prototype verification system using this technique for computers compliant with the Open Firmware boot standard.

Our verification technique is based on *Efficient Code Certification (ECC)* proposed in [6]. ECC is related to other recent language-based approaches to the security of mobile code [12, 15]. Each time an untrusted firmware module is loaded, it is verified against a standard security policy. Inexpensive static checks on the compiled code suffice to guarantee dynamic properties of the program. Among other things, the security policy asserts that device drivers must access other devices only through a strict interface and must only access memory or bus addresses allocated to them.

ECC verification relies on a certifying compiler that produces particularly well-structured and annotated code, so that the verifier can analyze it statically. The verification step essentially prevents the compiler from being bypassed, spoofed, or counterfeited. Confidence in the safety of verified device drivers only requires trust in the verifier, not in the compiler nor the code it produces. By “trust” here we mean that the user must have some other rational basis for believing in the integrity and correctness of the verifier – that it is in the *trusted computing base (TCB)*. The compiler and its output, on the other hand, do not have to be in the TCB. Any device driver code, whether produced by the compiler or not, must be verified.

This technique, while a strong countermeasure to malicious firmware, cannot protect against all forms of attack. For example, certain denial-of-service attacks and malicious hardware are difficult or impossible to detect by this method. However, it does raise the bar by making it more difficult to operate devices maliciously at boot time. Our approach is complementary to existing and proposed schemes that employ digital signatures, trusted suppliers, and code inspection. Those techniques would be appropriate to protect the integrity of the TCB, which will be relatively static.

Our prototype, currently under development, is compliant with the Open Firmware standard [5] and operates in that context. Open Firmware is an IEEE standard for boot firmware that was developed in the mid 1990’s and is by now in fairly widespread use (e.g., by Sun and Apple). Several commercial implementations are available. One key feature of Open Firmware that is responsible for its power and flexibility is its incorporation of boot-time device drivers and other modules written in *fcode*, a machine-independent compiled form of the Forth programming language. Open Firmware boot systems include an fcode interpreter, allowing a single implementation of fcode-based firmware to be reused across multiple platforms. The fcode driver is typically stored in ROM on the device itself and reloaded into main memory during the boot cycle. It is these fcode device drivers that are the subject of verification in our prototype.

The verifier is part of the Open Firmware boot kernel and is loaded from boot ROM when the machine powers up. The verifier, along with the fcode interpreter and other parts of the Open Firmware kernel, are considered part of the trusted computing base. The complementary protection schemes mentioned above will be appropriate for protection of this software because it is assumed to be static and supplied by a single vendor.

The security policy is a combination of type safety and various architectural constraints. The policy is designed to rule out the most obvious forms of attack. The constraints are a formalization of conventions that all legitimate fcode programs should adhere to, as well as restrictions that make verification easier without imposing undue limitations on the programmer. Those conventions are not strict requirements of Open Firmware, yet any firmware that violates them would likely be incorrect or malicious. For instance, each device driver conventionally operates its own device directly, and accesses the other devices only via their drivers.

A cornerstone of the ECC technique is a certifying compiler. Our prototype compiler translates Java Virtual Machine code (bytecode) to Forth fcode. We expect developers to use Java as the source language and compile to Java bytecode with a standard Java compiler such as the `javac` compiler from Sun Microsystems as the first stage. We are also developing a Java API so that these programs can access Open Firmware services and data structures. This API is not just a matter of convenience—it is a key element in our eventual assurance argument. The API presents a safer interface than the standard one; we will verify that untrusted code uses this interface and does not bypass it.

2 Open Firmware

Open Firmware is a standard for boot firmware platforms [5]. This standard enables vendors to write machine-independent and instruction set-independent boot firmware, including boot-time drivers. The major advantage to this approach is that Open Firmware-compliant firmware will work across a wide range of hardware. Sun Microsystems Open Boot works this way and was the inspiration for this standard.

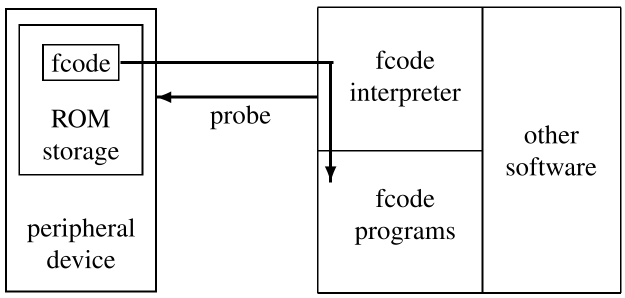

Figure 1. Fcode Loading in Open Firmware

Manufacturers of peripherals need only write one boot-time device driver. The same driver will work with any Open Firmware implementation on any platform. This driver is stored in ROM on the device itself.

The major tasks of boot firmware are:

- to determine the physical configuration of the host and peripherals and build the device tree data structure to represent this,
- to initialize those devices that require it, and
- to load the operating system (or runtime program) and start it running.

Open Firmware provides an abstract model of this process. A hardware-specific adaptation layer whose interface is defined in the standard supports this abstraction.

A key feature of Open Firmware is the incorporation of an interpreter for Forth fcode (Fig. 1). Forth is a stack-based programming language with a long history of use on microprocessors. Fcode is a standard compiled form of Forth that is very compact. Forth programs are called *words*, and a compiler that produces fcode from Forth is called a *tokenizer*. The mapping from Forth to fcode is completely defined in the Open Firmware standard.

Open Firmware boot systems contain an fcode interpreter. Such systems dynamically load and execute fcode modules during the boot cycle. Our system uses ECC-style verification, described in Section 3 below, to detect unsafe fcode programs.

Portions of the boot firmware (other than the adaptation layer) can be written in Forth and will run identically on different hardware platforms. This software will employ the standard boot data structures and hardware abstractions. In particular, peripheral devices are all accessed through a standard API consisting of a set of Forth words that each device of a particular type must define. The boot-time driver for each device is supplied in the form of an fcode program that when executed causes all required words to be defined appropriately. It also builds the portion of the device tree that represents this device. That fcode program is stored in ROM on the device itself. Open Firmware defines a standard method to retrieve the driver-defining code from any device. During the boot process, all of these programs are retrieved and executed, thus constructing an API for each device.

3 ECC

The ECC project (for Efficient Code Certification) [6] was conceived as a way to improve the runtime efficiency of small, untrusted, run-once applications such as applets and active messages while still ensuring safe execution. *Run-once* means that the cost of verification cannot be amortized over the lifetime of the code, so certificates should be as concise and easy to verify as possible.

ECC guarantees certain dynamic safety properties of compiled code by performing efficient static checks. In particular, it permits implementation of a module that, at boot-time, verifies the safety of the boot firmware before it is run. This technique relies on certain general mathematical theorems that relate the control flow safety, memory safety, and stack safety of a running program to the block structure of its compiled form. As a practical matter, the technique relies on a certifying compiler that produces particularly well-structured code, so that a verifier can perform appropriate static checks just prior to runtime. The user need only trust the verifier, which is a particularly simple program that can be persuasively validated by inspection.

ECC attempts to identify the minimum information necessary to ensure a basic but nontrivial level of code safety and to encapsulate this information in a succinct certificate that is easy to produce and to verify. Performance and ease of implementation are important concerns. ECC is able to ensure

- control flow safety—the program never jumps to a random location in the address space, but only addresses within its own code segment containing valid instructions;
- memory safety—the program does not access random places in memory, but only valid locations in its own data segment, system heap memory explicitly allocated to it, or valid stack frames; and
- stack safety—the state of the stack is preserved across subroutine calls.

These safety conditions are mutually dependent in the sense that none of them are safe unless all of them are safe. This level of safety is roughly comparable to that provided by Java bytecode verification. It also entails other ancillary safety properties such as checking the number and types of function call arguments.

A prototype certifying compiler for the Scheme language to Intel Architecture (x86) machine code and a corresponding verifier have been developed [6].

The system does not rely on general theorem provers or typing mechanisms. Although less flexible than other language-based approaches such as PCC or TAL [18, 12], certificates are compact and easy to produce and to verify. The certificate can be produced by the code supplier during the code generation phase of compilation and verified by the consumer at load time. Both operations can be made automatic and invisible to both parties.

Drawbacks to ECC include platform-dependence and fragility with respect to compiler optimization. Simple local optimizations such as tail recursion elimination can be handled. Preliminary experiments indicate that the sizes of the certificates produced by the ECC prototype range from 6% to 25% of the size of the object code. This seems to indicate a substantial improvement over PCC, although a fair comparison would require a more careful analysis to take all variables into account. The verification process is very efficient. It is linear time except for a sorting step to sort jump destinations, but since almost all jumps are forward and local, a simple insertion sort suffices.

4. The BootSafe System

In this section we describe in some detail our prototype, called *BootSafe*. We hope to convince the reader, as we are ourselves convinced, that this is a sound and commercially viable approach to protection against an important class of malicious boot firmware.

Our objective is to detect malicious fcode programs during the Open Firmware boot cycle as they are loaded, in order to prevent them from executing.

Detection is based on ECC verification as described in Section 3 above. However, verification will go beyond the basic safety properties of the original ECC prototype. This will require a meaningful security policy for fcode programs—essentially a conservative definition of program safety—and a means of enforcing that policy. Devising an effective policy is difficult because it is hard to foresee all the ways that an attacker could harm us.

The BootSafe system consists of a Java-to-fcode certifying compiler J2F and a corresponding fcode verifier (Fig. 2).

The following sections detail our approach to compilation and verification. Following that are sections containing some background information.

4.1 Compilation

The compilation of a Java program to fcode is a two-stage process. In the first stage, Java source is compiled down to Java Virtual Machine (VM) code, also known as bytecode. For this purpose we can use any existing Java compiler, such as the javac compiler available from Sun Microsystems. For the second stage, we are implementing a compiler J2F that maps Java VM code to Forth VM code, also known as fcode. Thus we can leverage existing compilers for Java. In addition, we will be able to leverage the Java bytecode verifier, as explained below.

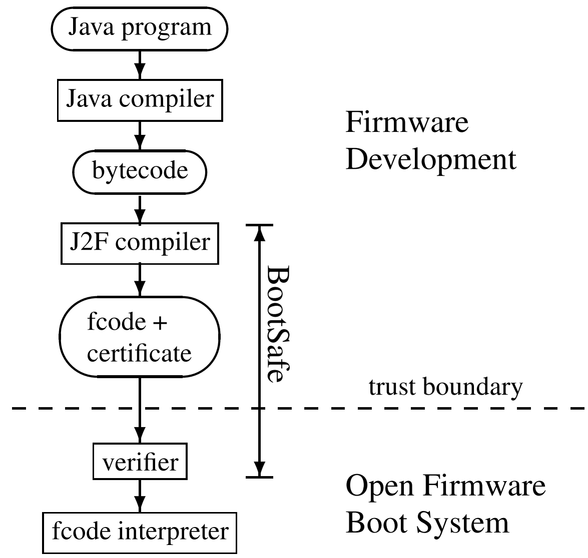

Figure 2. The BootSafe System

The translation from Java VM to Forth VM is relatively straightforward in many ways, although there are some design challenges. One such challenge is to design an appropriate object encoding and method dispatch mechanism. Since Forth contains no prior support for objects, we must design the object encoding from scratch. This goal has already been achieved.

Another issue is the class loading and initialization strategy. The standard Java strategy loads and initializes classes at their first active use. This is known as lazy initialization. For applications in boot firmware, lazy initialization is less desirable because it imposes a runtime overhead that could play havoc with real-time constraints that boot-time drivers are often subject to during diagnostic testing of devices. We thus prefer an eager initialization strategy to avoid this overhead. We have designed and implemented a load-time static analysis method that computes initialization dependencies among Java classes based on the static call graph of the class initializers [7]. We can use the computed dependencies to order the classes for initialization at load time. This is a departure from standard Java semantics, but a valuable one for this application. In addition to avoiding the runtime overhead of lazy initialization, it gives a clearer picture of the dependencies involved in class initialization, and can flag a variety of erroneous circularities that the standard lazy method lets pass.

Our J2F compiler can currently compile with int, boolean, and char types, String and StringBuffer classes, user-defined classes, and arrays of these types. It performs both static and instance method dispatch correctly. It incorporates our eager class initialization strategy as described in the preceding paragraph. The output of our J2F compiler is a working Forth program. The Forth code produced, along with a runtime support module of our design, can be successfully read and interpreted by the Forth interpreter that is part of SmartFirmware (a commercial implementation of Open Firmware). In the future we will utilize a backend mapping to fcode, and all verification will be done on the fcode, allowing us to produce a much more compact object involving Forth execution tokens instead of text. However, throughout the prototyping phase, we will continue to work with Forth source code so as to make it easier to debug the design of our J2F compiler and BootSafe verifier.

In the near future we will fill out the existing J2F prototype to handle a variety of other built-in types and library classes, eventually moving toward a significant and robust subset of Java, perhaps full Java minus only support for reflection and concurrency. It is also not clear that it will be necessary to support the double-word types long and double for applications in firmware. The elimination of these two types would result in a substantial simplification, since they requires special handling.

One significant advantage of Java over Forth is that the Open Firmware architecture is naturally object-oriented. Objects and inheritance can be used to great advantage in modeling the device tree and the various devices that comprise it. For example, the standard API of a PCI bus can be represented as an abstract class that manufacturers can subclass in order to implement drivers for their specific products. In fact, we will *require* such subclassing as one of the architectural constraints, mentioned above, that lead to enforcement of the security policy.

4.2 Verification

Sun has defined a form of safety verification for JVM bytecode as part of its definition of the JVM. Our verification will build on this, verifying analogous properties of fcode programs that have been translated from bytecode, as well as some new checks that are peculiar to the Open Firmware environment.

Verification consists of a general device-independent part that applies to all drivers and a device-specific part that may vary depending on the kind of device. The overall security policy we will eventually enforce is three-tiered. Enforcement of each tier depends on the previous ones.

Tier 1: Basic safety policy. This basic level corresponds to that which is commonly called type-safety in the literature on language-based security. It has a fixed platform- and application-independent description involving memory safety, control flow safety, and stack safety that are all interrelated. This level corresponds roughly to the level of safety provided by the Java bytecode verifier. It is also the easiest of the three levels to design and implement, because we can leverage the design of the Java bytecode verifier in our fcode verifier. Since we are translating from Java, it is possible to mimic the behavior of the Java bytecode verifier fairly directly, supplying the necessary typing information in the form of an attached certificate, as in the ECC prototype. Since we have a formal description of the Java verification process, we can supply in the certificate whatever extra information we need that may be present in the Java bytecode, then build our verifier to perform the same checks as the Java verifier. Thus the established type safety of Java and the existence of well-documented bytecode verification algorithms are a huge advantage that will save us much design time.

Tier 2: Device encapsulation policy. Each peripheral device is operated directly or indirectly only by its own device driver. Each device driver provides the only interface (API) for the rest of Open Firmware to access the corresponding device. Drivers that must access their devices through a chain of other devices, such as buses, must do so in a highly controlled and prespecified manner that is verifiable. Such forms of indirect access typically involve some form of address translation that is set up by the Open Firmware *mapin* procedure whose application can be tightly controlled. Although there is no software mediation at the time of the actual device access, it is possible for the verifier to check that the mapin procedure is called according to a highly constrained and recognizable pattern and that all bus addresses subsequently accessed by the driver are within the memory range allocated to that device by mapin. This is more or less comparable to an array bounds check in Java.

Tier 3: Prevention of specific forms of harm. In this tier, we enforce conventions that any reasonable device driver should adhere to. In so doing, we rule out many of the routes by which malicious fcode could cause harm. For instance, device drivers, once they are loaded, should never be redefined—there is no legitimate reason to do so. Such redefinition is otherwise legal within fcode, and would be a very attractive attack mechanism.

Enforcement of tier 2 and 3 policies will be based on architectural constraints—restricting the interaction of modules with one another and constraining the interfaces. This enables two strategies for enhancing safety: ruling out interactions that should never happen in legitimate firmware, and require that services are accessed via wrappers that perform run-time checks. Some Java language features such as private, protected, and final methods can be of great benefit here. For instance, by requiring that untrusted code is a subclass of a trusted class, we can ensure that final methods of the trusted class cannot be overridden in the subclass.

5. Related Work

In this section we discuss the mechanisms, such as cryptography and mediated access, that have been most commonly used to address security threats and their practical limitations for guarding against malicious firmware. We then discuss other important examples of language-based security mechanisms (Java, proof carrying code, and typed assembly language) and the tradeoffs involved in deciding which is appropriate for a particular application.

5.1 Non-Language-Based Mechanisms

Traditional approaches to the security problem include: mediated access or proxy execution, cryptography, code instrumentation, and trusted compilation.

Mediated access or proxy execution, probably the oldest and most widespread system security mechanism, proceeds by isolating critical operations and data in a system kernel, the only code privileged to access these operations and data directly. All other processes gain access only by using the kernel as a proxy, after communicating their desires by message. This not only prevents untrusted code from corrupting the system, but also allows the kernel to monitor all access, perform authentication, or enforce other safety policies.

Cryptography discourages access to sensitive data during transit across an untrusted network and can also be used for authentication.

Code instrumentation, software fault isolation (SFI), and sandboxing [24] alter (instrument) machine code so that critical operations can be monitored during execution in order to enforce a security policy. The monitor is invisible (except for performance costs) so long as execution follows the policy, but intervenes to protect the system when a violation is detected. Schneider [20, 21] extends this idea to handle any security policy that can be expressed by a finite-state automaton. For example, one can express the condition, “No message is ever sent out on the net after a disk read,” with a two-state automaton. These automata are called *security automata*. The code is instrumented so that every instruction that could potentially affect the state of the security automaton is preceded by a call to the automaton. Security automata give considerable flexibility in the specification of safety policies and allow the construction of specialized policies tailored to a consumer’s particular needs. The main drawback is that some runtime overhead is incurred for the runtime calls to simulate the automaton.

Trusted compilation is the practice of compiling locally using a trusted compiler.

None of these mechanisms are well suited to firmware. Firmware typically runs before the system kernel is even loaded. Mediation can be provided only by the BIOS, which operates in a relatively austere environment unequipped for proxy execution: firmware drivers associated with installed components are typically given direct access to critical system components. It is desirable to allow this capability without compromising security.

Code instrumentation is also costly. The runtime check required before every sensitive operation could contribute substantially to runtime overhead. Some runtime checks can be eliminated if program analysis determines that they are unnecessary, but this is also a costly undertaking and could contribute substantially to load time overhead. Moreover, even the most sophisticated analysis techniques are necessarily incomplete, because safety properties are undecidable in general.

Trusted compilation of the firmware would have to be redone every time a system is booted, incurring not only a performance penalty but the additional complexity of including the compiler in the trusted computing base. Also, trusted compilation does not by itself supply any justification for trust in the source code that is being compiled.

5.1.1 Cryptographic Authentication (AEGIS)

We have already noted that authentication alone cannot ensure that untrusted code is safe to run. Clearly, however, it can provide some protection. The most sophisticated authentication architecture for firmware is provided by AEGIS [2]. The prototype has been designed as a minimal change to the boot process of the IBM PC that provides a layered sequence of authentication checks of untrusted BIOS code and CMOS, then expansions cards, then the operating system boot blocks, etc., throughout the boot process. It also provides a mechanism for attempting to recover from a failed integrity check by obtaining a replacement module from a trusted source.

5.2 Language-Based Mechanisms

Compilers for high-level programming languages typically accumulate much information about a program during the compilation process. This information may take the form of type information or other constraints on the values of variables, structural information, or naming information. This information may be obtained through parsing or program analysis and may be used to perform optimizations or to check type correctness. After a successful compilation, compilers traditionally throw this extra information away, leaving a monolithic sequence of instructions with no apparent structure or discernible properties.

Some of this extra information may have implications regarding the safety of the compiled object code. For example, programs written in type-safe languages must typecheck successfully before they will compile, and assuming that the compiler is correct, any object code compiled from a successfully typechecked source program should be memory-safe. If a code consumer only had access to the extra information known to the compiler when the program was compiled, it might be easier to determine whether the code is safe to run.

Code certification refers to the idea of retaining extra information from a program written in a high-level language in the object code compiled from it. This extra information—called a certificate—is created at compile time and packaged with the object code. When the code is downloaded, the certificate is downloaded along with it. The consumer can then run a verifier, which inspects the code and the certificate to verify compliance with a security policy. If it passes the test, then the code is safe to run. The verifier is part of the consumer's trusted computing base; the compiler, the compiled code, and the certificate need not be.

5.2.1 Java

Perhaps the first large-scale practical instance of the language-based approach was the Java programming language [8]. Javas language-based mechanism is designed to protect against malicious applets. The Java runtime environment contains a bytecode verifier that is supposed to ensure the basic properties of memory, control flow, and type safety. In addition, a trusted security manager enforces higher-level safety policies such as restricted disk I/O. The Java compiler produces platform-independent virtual machine instructions or bytecode that can be verified by the consumer before execution. The bytecode is then either interpreted by a Java virtual machine (VM) interpreter or further compiled down to native code.

Early versions of Java contained a number of highly publicized security flaws [3]. For example, a subtle defect in the Java type system allowed an applet to create and control a partially instantiated class loader. The applet could then use this class loader to load, say, a malicious security manager that would permit unlimited disk access.

Some authors [3, 9] blamed these problems on the lack of an adequate semantic model for Java. Steps to remedy this situation have since been taken [1, 19]. Despite these initial failings, the basic approach was a significant step forward in practical programming language security. It not only pointed the way toward a simple and effective means of providing a basic level of security, but also helped direct the attention of the programming language and verification community to critical security issues resulting from the rise of the Internet.

The machine-independent bytecode produced by the Java compiler is still quite high-level, and that is a disadvantage. Once downloaded, the bytecode must either be interpreted by a Java VM interpreter or compiled to native code by a just-in-time (JIT) compiler. Either technique incurs a runtime penalty. If the safety certificate represented in the bytecode were mapped down to the level of native code by a back-end Java VM compiler, then the same degree of safety could be ensured without the runtime penalty, because the code supplier could do the back-end compilation before downloading. This would trade the platform independence of Java VM for the efficiency of native code.

5.2.2 Proof Carrying Code (PCC)

Proof carrying code (PCC) [13, 14, 15, 16, 17, 18] is a strategy for producing and verifying formal proofs that code meets general security policies. The software supplier does the hard work of generating the proof, and the software consumer checks the proof before the code is run. The security policy is expressed in first-order logic augmented with symbols for various language and machine constructs.

The most general version of PCC is somewhat more complicated, involving a two-phase interaction between the supplier and the consumer. In the first phase of this protocol, the supplier produces and delivers a program consisting of annotated object code. The annotations consist of loop invariants and function pre- and post-conditions, and make the next phase of the protocol easier. The consumer formulates a safety policy and uses an automated tool to generate, from the policy and the annotated program, a *verification condition*. The verification condition is a logical formula that implies that the program satisfies its security policy. In the second phase of the protocol, the supplier proves the verification condition and sends the proof back to the consumer. The consumer runs a proof checker to check that the proof is valid.

The verification condition generator is part of the consumer's trusted computing base—in a sense, it *defines* the security policy—but some communication cost can be saved by having the supplier generate the verification condition using the same verification condition generator that the consumer uses. The consumer then checks that the verification condition produced by the supplier is the same as the one produced locally.

A certifying compiler produces the initial annotation of the code, using information from the program source and program analysis during compilation. The Touchstone compiler [14] is a certifying compiler for a type-safe subset of C. It admits many common optimizations such as dead code elimination, common subexpression elimination, copy propagation, instruction scheduling, register allocation, loop invariant hoisting, redundant code elimination, and the elimination of array bounds checks.

The advantages of the PCC approach are its expressiveness and its ability to handle code optimizations. In principle, any security policy that can be constructed by a verification condition generator and expressed as a first-order verification condition can be handled. The main disadvantages are that it is a two-phase protocol, that it involves weighty machinery such as a full-fledged first-order theorem prover and proof checker, and that proof sizes are quite large, roughly 2.5 times the size of the object code for type safety and even larger for more complicated safety policies. Given the limited space available on boot firmware, this size penalty alone makes PCC inappropriate for our problem.

5.2.3 Typed Assembly Language (TAL)

Typed assembly language (TAL) [4, 9, 10, 12] can be viewed as a specialized form of proof-carrying code devoted to verifying a form of type safety. It is a language-based system in which type information from a strongly-typed high-level language is preserved as compilation transforms the source through a platform-independent typed intermediate language (TIL) [11, 23] down to the level of the object code itself. The result is a type annotation of the object code that can be checked by an ordinary type checker. In this special case, proof checking is reduced to type checking.

TAL is not as expressive as PCC, but it can handle any security policy expressible in terms of the type system. This includes memory, control flow, and type safety, among others. TAL is also robust with respect to compiler optimizations, since type annotations can be transformed along with the code. TAL proofs, though much smaller than proofs in PCC, are still significantly larger than those needed by ECC.

Proof size can be traded off against the complexity of the verifier, but that increases and complicates the amount of trusted code.

6. Current Project Status

The long-term goal of this project is to adapt the ECC technique to the analysis of Open Firmware fcode programs to detect malicious boot software. We are implementing a certifying compiler and verifier necessary for this method. The ECC-based verifier will then be incorporated into an existing commercial implementation of the Open Firmware standard in order to provide practical malicious boot firmware detection to the marketplace.

At present, we have a working prototype of the J2F compiler for Java Virtual Machine (JVM) bytecode to Forth fcode for a single-threaded subset of the Java language. This subset is appropriate for writing device drivers and other firmware modules. It provides access to Open Firmware services through an API currently being designed. The compiler will output a certificate appropriate to the verification tasks described below. The API takes advantage of the natural object-oriented structure of the Open Firmware device tree and allows access to Open Firmware services from within Java programs.

The BootSafe verifier will initially verify only basic type safety, roughly at the level provided by ECC and by the Java bytecode verifier. This initial version operates as a standalone program.

We have successfully compiled sample device drivers for a block-oriented storage device and a PCI bus. These are good representatives of typical devices in current use. These drivers can run under SmartFirmware (a commercial Open Firmware implementation) in simulation mode.

7. Conclusions and Future Work

As noted, typical boot firmware is an amalgam of many pieces, including libraries, the main boot program, and boot-time device drivers from various vendors. To interact successfully, these pieces must respect well-defined abstraction boundaries and communicate only via standardized interfaces. Yet at boot time, the pieces all run in the same address space. There is no isolation and no external enforcement of good citizenship. The existing Open Firmware standard does not address this problem. It only helps non-malicious designers by defining the standard for device interaction and process management during bootup.

Our approach has the potential to guarantee that all of the pieces of boot firmware are good citizens: that they respect each other's boundaries and interact only via published standardized interfaces. Moreover, this guarantee is refreshed each time the boot program runs with inexpensive static checks. Rechecking each time counters the threat of substituting malicious boot firmware components for approved ones.

We believe Open Firmware is the right context because it is a clear, well-designed, and widely used standard. We have designed a Java-to-fcode certifying compiler and built an early prototype. Our current effort is directed toward making this novel form of protection a practical reality by integrating the verifier with a commercial implementation of Open Firmware.

Although we our developing our techniques in the context of the Open Firmware standard, there is nothing to prevent non-Open Firmware compliant boot firmware from being made verifiable using similar techniques.

Among the large-scale issues still to be addressed are:

- the design of a Java API for Open Firmware that is both convenient to use and supports the kind of verification that we require;
- enhancement of the verifier to verify compliance of fcode programs with the second-order security policy (this version of the verifier will run as a stand-alone program and will be directly integrated with SmartFirmware);
- modification of the J2F compiler to accommodate the refined Open Firmware API and enhanced verification.

Acknowledgments

We are indebted to T. J. Merritt for valuable ideas and comments and to David Baca and Kori Oliver for their assistence with the implementation. We also thank the anonymous reviewers for their suggestions. This work was supported in part by DARPA contracts DAAH01-02-C-R080 and DAAH01-01-C-R026, NSF grant CCR-0105586, and ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the US Government.

References

[1] M. Abadi and R. Stata. A type system for Java bytecode subroutines. In *Proc. 25th Symp. Principles of Programming Languages*, pages 149–160. ACM SIGPLAN/SIGACT, January 1998.

[2] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *Proc. 1997 Symposium on Security and Privacy*, pages 65–71. IEEE, May 1997.

[3] Drew Dean, Ed Felten, and Dan Wallach. JAVA security: From HotJava to Netscape and beyond. In *Proc. Symp. Security and Privacy*. IEEE, May 1996.

[4] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Proc. 26th Symp. Principles of Programming Languages*, pages 250–261. ACM SIGPLAN/SIGACT, January 1999.

[5] IEEE. *IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices*, 1994. Standard 1275-1994.

[6] Dexter Kozen. Efficient code certification. Technical Report 98-1661, Computer Science Department, Cornell University, January 1998.

[7] Dexter Kozen and Matt Stillerman. Eager class initialization for Java. In W. Damm and E.R. Olderog, editors, *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02)*, volume 2469 of *Lecture Notes in Computer Science*, pages 71–80. IFIP, Springer-Verlag, Sept. 2002.

[8] Tim Lindholm and Frank Yellin. *The JAVA virtual machine specification*. Addison Wesley, 1996.

[9] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Proc. Workshop on Compiler Support for System Software*, pages 25–35. ACM SIGPLAN, May 1999.

[10] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In Xavier Leroy and Atsushi Ohori, editors, *Proc. Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52. Springer-Verlag, March 1998.

[11] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *1996 Workshop on Compiler Support for Systems Software*, 1996.

[12] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *25th ACM SIGPLAN/SIGSIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego California, USA, January 1998.

[13] George C. Necula. Proof-carrying code. In *Proc. 24th Symp. Principles of Programming Languages*, pages 106–119. ACM SIGPLAN/SIGACT, January 1997.

[14] George C. Necula. *Compiling with proofs*. PhD thesis, Carnegie Mellon University, September 1998.

[15] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd Symp. Operating System Design and Implementation*. ACM, October 1996.

[16] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proc. Conf. Programming Language Design and Implementation*, pages 333–344. ACM SIGPLAN, 1998.

[17] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Proc. 13th Symp. Logic in Computer Science*, pages 93–104. IEEE, June 1998.

[18] George C. Necula and Peter Lee. Safe, untrusted agents using using proof-carrying code. In Giovanni Vigna, editor, *Special Issue on Mobile Agent Security*, volume 1419 of *Lect. Notes in Computer Science*, pages 61–91. Springer-Verlag, June 1998.

[19] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th Symp. Principles of Programming Languages*, pages 70–78. ACM SIGPLAN/SIGACT, January 1999.

[20] Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proc. 11th Int. Workshop WDAG '97*, volume 1320 of *Lecture Notes in Computer Science*, pages 1–14. ACM SIGPLAN, Springer-Verlag, September 1997.

[21] Fred B. Schneider. Enforceable security policies. Technical Report TR98-1664, Computer Science Department, Cornell University, January 1998.

[22] Fred B. Schneider, editor. *Trust in Cyberspace*. Committee on Information Systems Trustworthiness, Computer Science and Telecommunications Board, National Research Council. National Academy Press, 1999.

[23] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Conf. Programming Language Design and Implementation*. ACM SIGPLAN, 1996.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L Graham. Efficient software-based fault isolation. In *Proc. 14th Symp. Operating System Principles*, pages 203–216. ACM, December 1993.