Intermediate Representations of Mobile Code

Wolfram Amme and Thomas S. Heinze
Friedrich-Schiller-Universität Jena, Germany
E-mail: {amme,theinze}@informatik.uni-jena.de

Jeffery von Ronne
The University of Texas at San Antonio, USA
E-mail: vonronne@cs.utsa.edu

Overview paper

Keywords: mobile code, intermediate representation

Received: April 3, 2007

Over the past decade, since Java was first introduced and integrated into the Netscape web browser, several intermediate representations have been developed that might be potentially used for mobile code applications. This paper examines the requirements for a mobile code representation, presents several examples of stack-based, tree-oriented, and proof-annotating mobile code representations, and evaluates each of these representations according to the requirements.

Povzetek: Članek podaja pregled mobilnih kod.

1 Introduction

In this era of the Internet, we increasingly come across mobile code applications (i.e., programs that can be sent in a single form to a heterogeneous collection of processors and will then be executed on each of them with the same semantics [1]). Such mobile code is usually intended to be loaded across a network and executed by an interpreter or after dynamic compilation on the target machine.

Unlike traditional monolithic, statically-compiled applications, many modern applications are designed to be dynamically composed from or extended with new components at runtime. An example of this is the Eclipse software development platform [17] that allows new plugins written in Java to be integrated into the environment. This dynamic extensibility is enhanced when the plugins can be described by executable code deployed in a mobile code representation that has a greater compactness, portability, and safety than native binaries.

The Java Virtual Machine's bytecode format ("Java Bytecode") has become the de facto standard for transporting mobile code across the Internet. However, in the last decades several intermediate representations of mobile code have been developed, each of which could be used as an alternative to Java Bytecode. In the paper we give an overview of common intermediate representations, discuss the strengths and weaknesses of each, and finally compare its attributes with that of the other representations.

The intermediate representations designed for mobile code are complex and usually combine multiple features and mechanisms. Therefore, a clear classification of mobile code representations is awkward. In contrast to other surveys, in which mobile code is examined from a programming language perspective [69] and for their verification time [45], respectively, our categorization emphasizes the structure of the intermediate representation. In particular the overview in [69] focuses on several programming languages (Java, Objective Caml, Telescript, etc.) and their suitability in mobile code environments. These languages are not classified by a taxonomy, but are introduced sequentially and evaluated according to some of the requirements imposed by the mobile code setting. In contrast, the article [45] centers on compiling safe mobile code, stressing the importance of safety in the mobile code setting. It discusses the safety issues present in several intermediate representations and compilation techniques, and classifies intermediate representations of mobile code according to their safety checking mechanisms, differentiating static, dynamic and hybrid mechanisms. The static mechanisms check critical safety properties at compile time (e.g., by static program analysis), while dynamic mechanisms rely on runtime safety checks (e.g., by inserting runtime checks into the code). Hybrid mechanisms apply a combination of static and runtime safety checks. In contrast, our classification does not focus on a single aspect (like safety) but highlights the general design of intermediate representations used for mobile code and classifies them as stack-based, tree-oriented, or proof-annotated.

The paper is structured as follows: In Section 2, we introduce a general framework for program transport by means of mobile code, and specify requirements this imposes on intermediate representations of mobile code. Section 3 presents our taxonomy and lists the primary representatives of each category. An evaluation and comparison of these intermediate representations is given in Section 4, and Section 5 concludes with a summary and a discussion about future directions in the area of mobile code representations.

2 Mobile code and its requirements

A system for transporting programs as mobile code can be partitioned into a producer side and a consumer side (see Figure 1). These two components communicate through files containing the mobile code in some intermediate representation (IR). The first step on the producer side is to analyze the input program syntactically and semantically and to transform it into an abstract syntax tree (AST). In the next stage, platform-independent optimizations can be performed, and annotations supporting consumer-side program analysis and optimization may be added to the abstract syntax tree in order to speed up dynamic code generation. Finally, the program is transformed into the chosen intermediate representation, and after being encoded as a—possibly compressed—binary, they are stored into files.

These files containing mobile code are then transferred to the consumer side where they are decoded. Next, the transmitted program has to be examined to determine if it adheres to the security requirements of the mobile code format. This verification process can use a variety of mechanisms ranging from simple type checks to validation of digital signatures or even the verification of proofs about program properties. If no violations are found, the program is executed on the target machine. The execution environment can execute the program by interpreting it or using a just-in-time (JIT) compiler to generate native machine code that runs directly on the target machine. In order to improve performance, JIT compilers often perform machine-dependent optimizations on the program code; this consumer-side optimization is sometimes enhanced by producer-side program annotations. To fulfill the requirements of a mobile code framework, special attention needs to be paid to the choice of intermediate representation. A candidate intermediate representation of mobile code can be evaluated on its ability to satisfy several desirable properties [25]:

Portability: An important property of an intermediate representation of mobile code is high portability. The mobile code needs to be able to execute on different target platforms, so the intermediate representation must be independent from any specific target machine's architecture.

Compactness An intermediate representation should also be dense. Originally, this requirement was due to restricted memory on some of the target code consumers, but today it is more important for reducing transmission times. This property is still critical, especially with respect to dynamically loaded mobile programs.

Flexibility: If an intermediate representation is not bounded to a specific input programming language, it can be used for a wide range of languages. This implies the advantage, as stated in [35], of implementing only *n* code-producers and *m* code-consumers instead of implementing *n*m* compilers. To attain high flexibility, the intermediate representation must support a versatile instruction set and an abstract type model.

Safety: In a mobile code system, the partitioning into code-producer and code-consumer leads to situations, in which the mobile code is not delivered directly by a trusted code-producer. Therefore, the question comes up as to how the code-consumer can ensure that the execution of the mobile code does not maliciously or accidentally affect the local machine in an unauthorized manner. Hence, an appropriate intermediate representation must support verification techniques to ascertain safety properties as type and memory safety.

Efficiency: Finally, although efficiency ultimately depends on the quality of the mobile code system implementation, the intermediate representation of a mobile code system should facilitate, or at least not hinder, the efficient execution of mobile code applications. This property is affected by the way the mobile code is executed: interpreted or compiled just in time. An interpreter-based implementation usually yields lower memory and other resource usage, and is often the most appropriate implementation for embedded systems. An implementation based on just-in-time compilation, however, usually results in faster execution of frequently executed mobile code and also supports machine-dependent program optimizations. Features that make a program representation easier to interpret may make it more difficult to optimize during JIT compilation or vice versa.

An optimal intermediate representation for mobile code should satisfy all of these properties, however in practice, the representation's designer may have to make design decisions that prioritize one over the other. As an example, increases in the safety guarantees often incur a loss of efficiency due to the increased costs of the verification process. Therefore, even though a representation cannot maximize all of these properties simultaneously, a mobile code representation can be evaluated on the basis of how well it satisfies these requirements.

3 Verifiable mobile code representations

Since mobile code is often received through untrusted channels, it is critically important to preserve the mo-

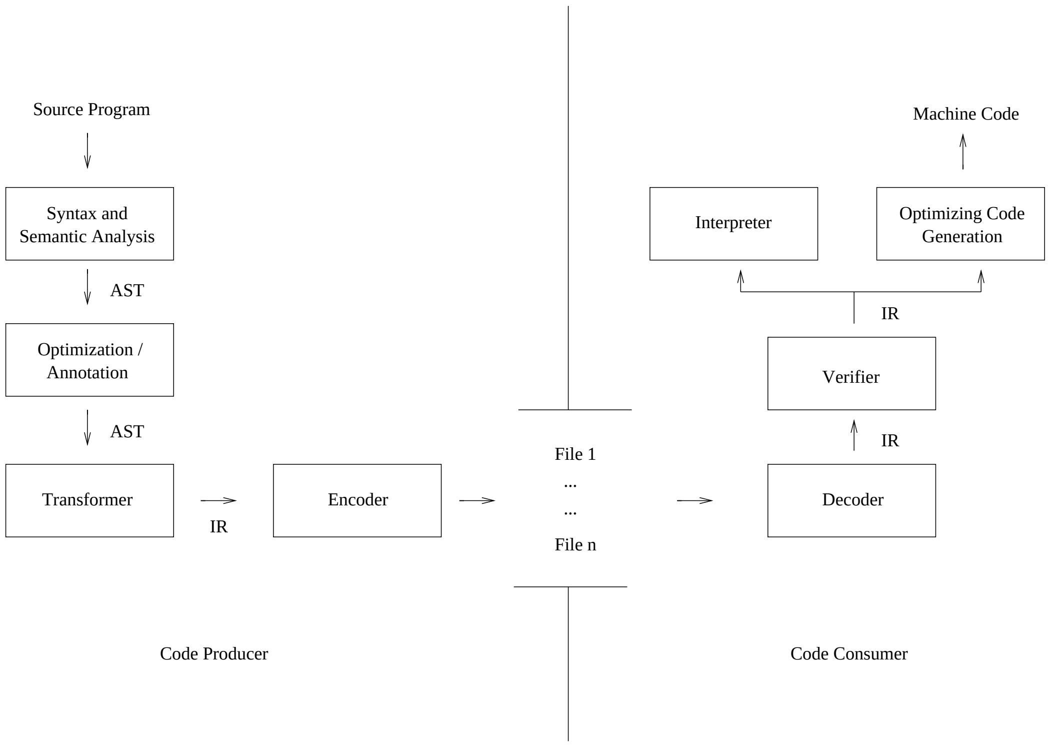


Figure 1: A general system for program transport by means of mobile code.

bile code consuming host system's security in the presence of malicious code. There are three main strategies that have been used—sometimes in isolation, sometimes in combination—to address this risk: cryptographic authentication, sand-boxing, and verification.

The first strategy is to use cryptographic signatures to authenticate the mobile code's producer and to prevent the mobile code from being tampered with during transit from the code producer. The code consuming system can then make decisions about the execution of the mobile code based on the trustworthiness of the code producer. In the simplest form (e.g., Microsoft's ActiveX Controls [53]), this may simply be used to run or not to run the mobile code. In more complex situations, this is used in combination with a security policy and "sand-boxing" to prevent mobile code from performing unauthorized actions.

A second strategy is to create a "sandbox" around the executing mobile code and mediate all access to parts of the code consuming system outside of the sandbox. This allows the sandbox to prohibit those interactions that violate the code consuming system's security policies. This sandboxing can be implemented using operating-system level (e.g., VMWare, Xen, User Mode Linux) or process-level isolation (e.g., BSD jail, SELinux), but these techniques are too heavy-weight and too loosely integrated for use in many mobile code applications (e.g., a Java applet running on a cell phone). These problems can be addressed by using light-weight, fine-grain isolation integrated into the execution environment.

A third strategy, often used to implement fine-grain isolation, is to analyze the mobile code and reject code that violates certain safety properties. Most commonly, this "verification" checks that the code is syntactically correct, has legal control flow, and that it is correctly typed. If the mobile code representation itself is type safe, this will guarantee the possible behavior (especially, with respect to memory accesses) of the mobile code to be constrained by the underlying mobile code representation's type system. This in turn will allow the execution environment (e.g., the Java Virtual Machine) to sandbox mobile code components without necessitating the runtime overhead of operating system or process level techniques.

Successful implementation of this third strategy requires that the program representation is designed with verification in mind. The verifiable mobile code representations that have been used in mobile code frameworks can be classified as being stack-based, tree-oriented, or proof-annotated representations.

3.1 Stack-based types

Most mobile code systems are based upon virtual machines. In such mobile code systems, programs are translated not into machine code for a specific target machine but rather into a platform independent intermediate representation. This intermediate representation consists of instructions for an idealized "virtual" machine.

Code consumers simulate the virtual machine by interpreting the transmitted intermediate representation or by

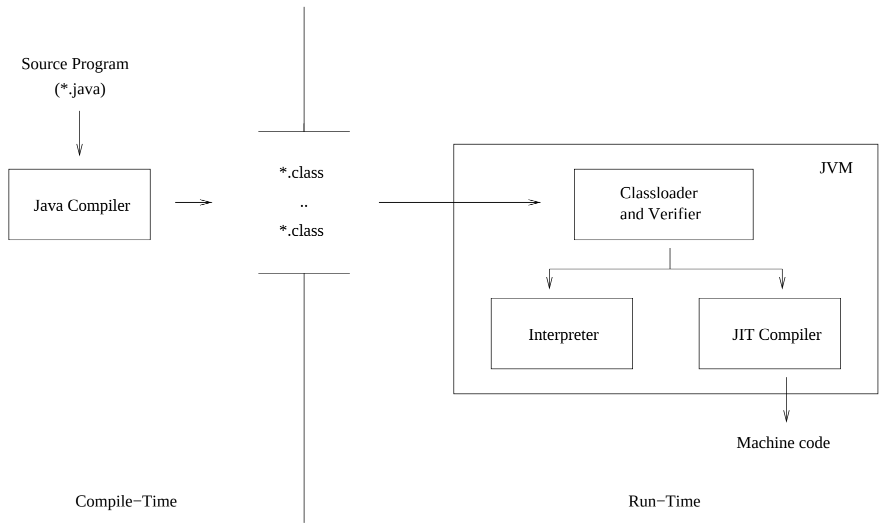


Figure 2: Java language infrastructure.

compiling it into equivalent machine code. Most often the virtual machine utilizes a stack-based architecture. In these virtual machines, most instructions implicitly take most of their operands from a stack and store most of their results back onto the same stack. One advantage of this architecture is the compact encoding of instructions; since most operands are implicit, many instructions can be represented by a single opcode without any operands. These representations are often designed so that each instruction can be encoded using a single byte, and for this reason the instruction sets of stack-based virtual machines are often called bytecode.

Virtual machines have long been used in compiler construction. Starting in the 1970's, compilers have used this concept to organize machine-dependent and machine-independent phases into front end and back end of a compiler. A representative example is P-Code [60], a stack-based intermediate representation used in some Pascal compilers. In the 1990's, *Sun MicroSystems* revived interest in stack-based virtual machines with the Java programming language and its portable intermediate representation, Java Bytecode [34, 33]. Microsoft's .NET Framework [63] also uses a stack-based intermediate representation called the Common Intermediate Language.

3.1.1 Java bytecode

Java Bytecode is a stack-based intermediate representation that was developed as type-safe program representation for Java programs. The instruction set and data types of Java's Virtual Machine (JVM)[1] are designed specifically for the Java programming language. In principle, Java Bytecode can also be used for other programming languages, but field reports show that a use of Java Bytecode for languages other than Java often can cause problems [23].

[1]A detailed description of JVM is given in [49].

The architecture for the typical deployment of Java as a mobile code system is given in Figure 2. On the producer-side of this system, a compiler translates a source program into portable Java Bytecode representing each Java method; all the methods in each class (with associated symbolic information) are stored together in a Java "class file." After successful transmission, the consumer-side JVM verifies the code to determine if it is safe to execute the mobile program. If the verification succeeds, the Java Bytecode is interpreted or executed directly after JIT compilation into machine code from the Java Bytecode.

The JVM's most important components are a runtime stack, a program counter, and heap storage, which store objects, code segments, and symbolic information. If a method is invoked, a new method frame is created and placed by the JVM onto the top of the runtime stack. This method frame contains the values of parameters and local variables as well as information about the caller. In addition, each frame contains an operand stack which is accessed as Java Bytecode instructions need input operands and produce output results. Each slot of the operand stack can hold a 32-bit word, and two slots are needed for long or double values.

Figure 3(b) depicts the Java Bytecode program generated for a simple source code. In the program, the contents of local variables a and b are added and its result afterwards is stored in local variable c. JVM assigns indices[2] to all parameters and local variables within a method. These indices are used instead of their symbolic names to reference local variables and parameters. The density of Java Bytecode is increased by the inclusion of instructions which implicitly encode the indices of the first variables defined in a method.

In the sample program, the instructions iload_1 and

[2]The index 0 is reserved for the object reference in case of virtual methods.

```
(a)
int a,b,c;
    c = a + b;

(b)

iload_1    ; push local variable a onto stack
iload_2    ; push local variable b onto stack
iadd       ; add topmost stack elements
istore_3   ; store topmost stack element into local variable c

(c)

ldloc.1    ; push local variable a onto stack
ldloc.2    ; push local variable b onto stack
add        ; add topmost stack elements
stloc.3    ; store topmost stack element into local variable c
```

Figure 3: Java Bytecode (b) and CIL Bytecode (c) for a simple program (a).

iload_2 are used to push the values of *a* and *b* onto the operand stack. In contrast, the instruction istore_3 takes the topmost element from the operand stack and stores its value in variable *c*. Most of the bytecode instructions are typed (i.e., only accept operands of a specific type). The operations that start with *i* generally indicate that they only accept values of type *int* as operands. Thus, in the example program, instruction *iadd* takes the two top-most int values from the operand stack, adds them, and stores the result back on the top of the operand stack.

The primary design consideration during the development of the JVM was its usefulness as a runtime environment for Java. Therefore, the JVM's instruction set is specialized for the representation of Java programs. Java Bytecode supports four different method invocation instructions implementing the virtual, super, static, and interface method calls of the Java programming language. For each method call, parameters are passed by value only, reference parameters are not supported directly. The JVM's flexibility with respect to running programs written in other languages is also limited by the JVM's provision of only single-inheritance for classes and multiple-inheritance for interfaces, respectively. Another disadvantage is the absence of arithmetic exceptions beside the division-by-zero exception for integers.

Java Bytecode's verification process includes static and dynamic checks and basically operates in four separate passes:

- Examination of general class file format
- Examination of additional structural properties of the class file
- Verification of the bytecode for each method
- Verification of inter-class dependencies during the execution of particular bytecode instructions

The examination of the transmitted Java Bytecode method (3rd pass) is performed by a data flow analysis, which verifies that certain behaviors, which might violate the virtual machine's type discipline (e.g., operand stack over- and underflows, unequal sizes of the operand stack on different control paths, usage of uninitialized local variables, operands of incorrect types for the operation) cannot occur. Overall, the data flow analysis is quite complex and requires, in the worst case, quadratic time in the number of verified instructions[66].

In contrast to the first three passes (which are performed during loading and linking process), the last verification pass, which checks properties about external classes referred to by bytecode instructions, occurs at runtime. In principle, all of the properties that are checked during this pass (e.g., that the classes referred to by an instruction exists) could be performed also during pass 3. But the JVM specification allows these checks to be deferred until run time, so that the loading of additional classes can be deferred until the instructions that refer to these additional classes need to be executed. If one of these dynamic check fails, the execution of the instruction being checked is aborted and an exception is thrown.

3.1.2 Common intermediate language

Microsoft Corporation's Common Language Infrastructure Platform (CLI) is a runtime environment that has been developed for running applications written in several different programming languages, including C#. CLI includes a stack-based virtual machine, called the Common Language Runtime (CLR), which can be used for execution of bytecode programs written in Common Intermediate Language (CIL). In contrast to JVM, the CLR standard does not anticipate execution with interpreter, but rather assumes all applications will be executed using JIT or ahead-of-time compilation.[3]

The .NET-Framework is Microsoft's proprietary implementation and extension of the CLI. In its current version,

[3]Mono, the CLR (ECMA-335) implementation from Novell, however, does include an interpreter [16].

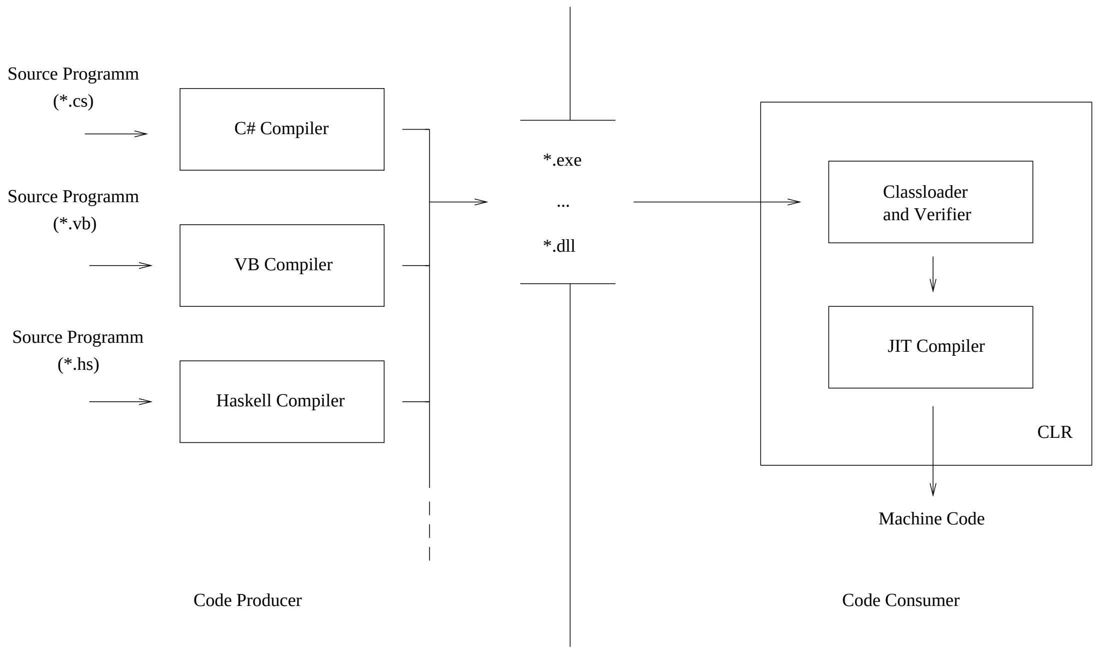


Figure 4: .NET framework.

.NET uses two JIT compilers: A standard JIT compiler and Econo. .NET's standard JIT compiler is an optimizing compiler that supports several optimizations (e.g., constant propagation, method-inlining, common subexpression elimination). In contrast, Econo is a non-optimizing JIT compiler that requires few system resources and, therefore, is especially suited for deployment on mobile platforms with limited resources. In addition, in the .NET-Framework, programs (or parts of programs) may be compiled in advance by the Pre-JIT compiler. Programs, that have been compiled with this compiler, are stored on the file system permanently, so that they can be executed directly when needed in the future without needing to be recompiled at runtime by the JIT compilation.

For each method invocation the CLR creates a new activation record. An activation record consists of fields containing method information, an instruction pointer, arrays for local variable and parameter definitions, and an evaluation stack. The stack-architecture of the Common Language Runtime is realized by the evaluation stack, which is used like the operand stack of the JVM to store the operands and the results of CIL instructions. In contrast to the JVM operand stack, the CLR evaluation stack is capable of storing elements of variable size.

Figure 3 (c) shows the CIL bytecode generated for the sample program from (a). Similar to local variable access in Java Bytecode, local variable and parameter accesses in CIL occur through indices assigned to variables and parameters in the order of their declaration. In the example program, instruction *ldloc* is used to push the value of a variable onto the evaluation stack. In contrast, instruction *stloc* takes the topmost element of the evaluation stack and stores it in a variable. For each variable, there is a corresponding load instruction and a corresponding store instruction to access that variable. These instructions are named by adding the variable number as a suffix (e.g. .1, .2, and .3,) to the operation name. In the example program, instruction *stloc.2* stores the topmost stack element in the local variable *b* (i.e., the local variable with associated index 2.) Unlike Java Bytecode, CIL offers the developer typed and untyped instructions. In the example program, the generic *add-operation* is used. Uses of this generic add-instruction require the CLR to infer the type of add-instructions during JIT compilation from its actual operand types.

In contrast to JVM, the CLI was developed with the intent of supporting many different programming languages. Therefore, the instruction set of CLR is designed around a general type system that is called Common Type System (CTS). Beside the standard primitive and reference types found in Java, the CTS also includes value types. A value type is essentially a restricted class, that is similar to a structure or enumeration type. Like the Java Virtual Machine, the CTS offers only single-inheritance of classes but multiple-inheritance of interfaces. The flexibility of this type model is further enhanced by the instruction set of the CLR, which includes several instructions to make the execution of programming languages other than C# more efficient. For example, a .tail suffix can be appended to a method call instruction, causing it to discard the stack frame of the calling method; this is particularly important for the efficient implementation of functional languages, which make heavy use of recursive calls that would otherwise overflow the runtime stack.

For method invocations there are two call instructions (*callvirt* and *call*) that can be used for virtual, non-virtual and static method calls. For parameter passing CLI offers call-by-reference and call-by-value mechanisms. In addition, parameters can be characterized as result parameters. Standard exception handling for operations on primitive data types are supported only for integer null-division. However, in contrast to JVM, in CLI add-, sub- and mult-instructions can be extended with special postfix operands to handle overflow exceptions.

When the producer side of the mobile code system translates source programs into CIL, it packages them into "assemblies." An assembly contains a set of modules bundled together along with meta-data describing the classes and types defined in and used by those modules [50]. In contrast to Java Bytecode's class files, which contain only a single Java class, a CIL assembly is able to contain several classes. This facilitates the composition of application programs out of multi-module components and allows the producer-side compiler greater scope for inter-class and inter-procedural optimizations. The code within the modules provides sequences of Common Intermediate Language instructions defining the behavior of the methods declared in the assembly.

The CLR uses a verification process, similar to that of the JVM, to determine if it is safe to execute CIL programs. Unlike the JVM, the CLR can be configured to allow certain programs to use "unmanaged" instructions, which can break the type safety of the runtime environment. These are provided in order to support a wide range of programming languages, including languages with unsafe features like pointer arithmetic. Normally, these unsafe instructions would be disabled when running mobile code.

The verification process is performed in two passes: validation and verification. In the validation pass, the general assembly format and the proper use of the meta-data format is ensured. Therefore, the validation pass corresponds to the first two passes of the Java Bytecode verification. In addition, a successful validation is a prerequisite for the verification pass, which is used to verify the control flow and then type-check the CIL module. This verification pass mirrors the last two passes of the Java Bytecode Verification and uses similar mechanisms.

3.2 Tree-oriented representations

Many compilers translate source programs into intermediate representations based on abstract syntax trees. Tree-oriented mobile code representations are derived from these internal data structures, linearized into a stream of binary data so that they can be transmitted in files or across the network. Due to their close relationship to internal compiler structures, tree-oriented intermediate representations are especially well-adapted to execution through JIT compilation, but they can also be interpreted.

A typical tree-oriented mobile code representations compilation unit consists of a source module's abstract syntax tree and symbol table of a program (which would typically be generated during the compilation of the source program even if native machine code were to be targeted) [12, 29, 39, 28]. Since abstract syntax trees are typically machine-independent, tree-oriented intermediate representations are often very portable. In addition, the semantic gap between source language and mobile code representation is minimized compared to a translation into stack-oriented bytecode [66]. The advantages of this approach include the retention of high-level program information (e.g., types and control structures), that can be useful for program optimizations, and a verification process that more closely resembles the type-checking of the source language. The primary disadvantage of this approach is that because it is closely tied to a single source language, it tends not to be very flexible with respect to supporting other source languages.

Though not a true mobile code representation (since it does not address network transportation or verifiability), the Architecture Neutral Distribution Format (ANDF) demonstrates the portability benefits of platform-independent tree-oriented program representations. The compact tree-oriented representation, Slim Binaries, demonstrated the viability of transporting mobile code applets over networks using a tree oriented rather than a stack-oriented representation (like Java Bytecode). The SafeTSA representation is a hybrid representation that combines tree-oriented control structures with blocks of instructions in static single assignment form, which is commonly used as an intermediate representation of the back end of optimizing compilers.

3.2.1 ANDF

The Open Software Foundation's Architecture Neutral Distribution Format (ANDF) [61] was a subset of the Ten15 Distribution Format (TDF)[4] developed by the Defense Research Agency in the UK (DRA). TDF [13] is a tree structured language, that is defined as a multi-sorted abstract algebra. It was originally designed for the compilation of sequential languages such as C and Lisp.

The intended usage was that programs would be distributed in the ANDF, and then compiled into native code at installation time. As such, ANDF was designed solely as a distribution format with a tree-oriented program representation that supports several source programming languages.

Inside of the ANDF infrastructure (see Figure 5), the producer-side translates a program to distribute into ANDF, expressing platform specific information by standard application programming interfaces (API's) [8]. Thereafter, the generated ANDF program is encoded into files and transmitted to the consumer-side, called installer. To install the transferred program, the ANDF files are compiled into target platform's machine code, and the installer replaces calls to an API with implementations provided by the target platform. Although originally developed for the C language,

[4]later renamed to the TenDRA Distribution Format.

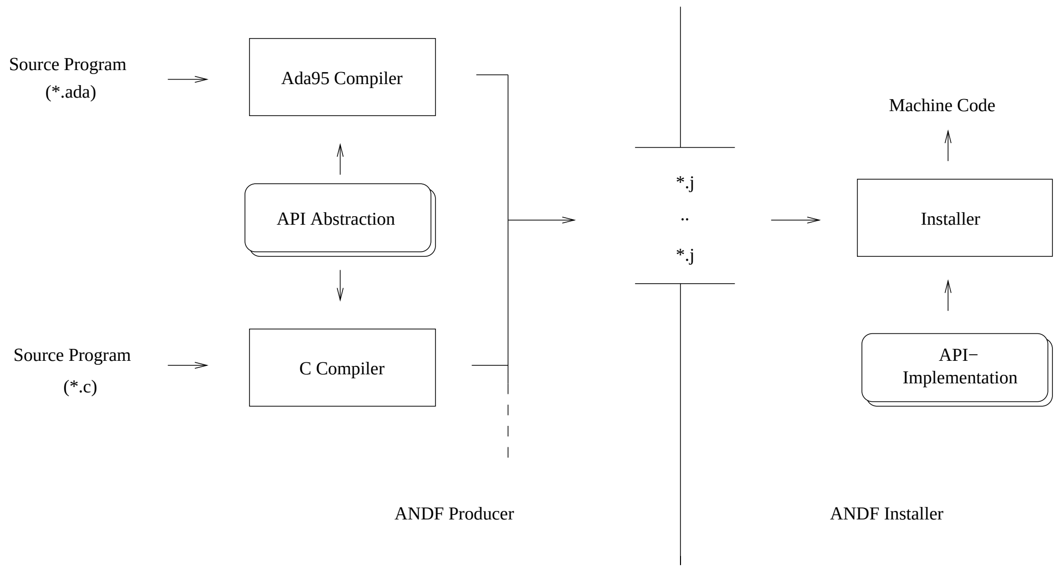


Figure 5: ANDF Scenario.

```
(a) int i,j;

       i = i + 1;
       j = j + 1;
       if (i <= j)
          i = i + 1;
       else
          i = i − 1;
       j = j + 1;

(b) sequence(
      assign(
         obtain_tag(~tag_1),
         plus(contents(integer(~signed_int*),obtain_tag(~tag_1)),
              make_int(~signed_int*,1))),
      assign(
         obtain_tag(~tag_2),
         plus(contents(integer(~signed_int*),obtain_tag(~tag_2)),
              make_int(~signed_int*,1))),
      conditional(
         ~label_0,
         sequence(
           integer_test(
              less_than_or_equal,
              ~label_0,
              contents(integer(~signed_int*),obtain_tag(~tag_1)),
              contents(integer(~signed_int*),obtain_tag(~tag_2))),
           assign(
              obtain_tag(~tag_1),
              plus(contents(integer(~signed_int*),obtain_tag(~tag_1)),
                   make_int(~signed_int*,1))),
           assign(
              obtain_tag(~tag_1),
              minus(contents(integer(~signed_int*),obtain_tag(~tag_1)),
                    make_int(~signed_int*,1))))),
      assign(
         obtain_tag(~tag_2),
         plus(contents(integer(~signed_int*),obtain_tag(~tag_2)),
              make_int(~signed_int*,1))))
```

Figure 6: A sample program (a) and its ANDF output (b).

ANDF producers and installers are available for other programming languages and several machine architectures [9].

In TDF, the original program structure is maintained within the intermediate representation. The base element of the TDF is the sort constructor. Instances of this constructor represent abstractions of expressions, descriptors, and data types. The shape constructor is used to describe data types within the TDF, including procedures, pointers, and recursive data types beside the primitive data types. Generic types can be defined in order to support platform specific data types, like the native integer type. Other sort constructors can be used to define specific memory layouts for data structures, exception handlers, and runtime stacks.

TDF includes various operations, which can be separated into arithmetic, memory, pointer, and control flow operations. Each operation is described using the expression constructor. Figure 6 (b) shows some simplified ANDF output for the example program given in (a). Descriptors (e.g., variables) in TDF are defined by the tag constructor. In this ANDF sequence, a unique integer is assigned to each tag constructor, in which *tag_1* stands for variable *i* and *tag_2* describes variable *j*.

Platform independence is achieved in TDF through the provision of two constructs: the token constructor and the conditional constructor. The token constructor is essentially a parameterized placeholder, which can be replaced with an arbitrary sort constructor. Therefore, the token constructor is used within TDF to hide platform specific program information by substituting calls to an API. In addition, the conditional variant of several constructors allows one to specify platform specific installation tasks. A conditional constructor includes two constructors and a condition: the installer evaluates the condition and maintains one of the constructors, corresponding to the result.

In general, the installation process of ANDF programs is separated into two steps. In the first step, calls to API's, denoted by token constructors, are replaced with its corresponding implementation. In the second step, conditional constructors of the program are evaluated. As a result, the platform independent ANDF program is transformed on the consumer-side into a platform dependent ANDF program, which then is compiled into the machine code of the target platform and installed.

For the transport of ANDF programs, the algebraic TDF is linearized and stored in a *capsule file*. A capsule file consists of a byte array structured into sections. The first section includes the definitions of visibility rules for the encoded ANDF program and acts as an interface. All of the token constructors used in the capsule are specified in the next section with the definitions of token constructors following their declarations in order to simplify the encoding and decoding process. After all the token constructors have been specified, the program is stored in the following sections using a linearized version of its TDF representation. A capsule file normally contains a single program, but it is possible to merge several capsule files into a single capsule library.

Verification of capsule files by the installer on the consumer-side is not integrated into the ANDF scenario, due to its development as a program distribution format. Instead, ANDF producers and installers are validated with respect to their conformity to the ANDF specification during a certification process [8]. For certification, ANDF producers and installers are validated separately. Validation of an installer is based on a number of hand-written programs (i.e., the ANDF Validation Suite [40]), which must be executed accurately by an ANDF installer. Validation of a producer is more difficult, because the produced ANDF code must execute correctly in any runtime environment for which there is an ANDF installer. Therefore, an architecture-independent high-level interpreter is used to evaluate the correctness of ANDF code generated by an ANDF producer for the ANDF Validation Suite.

3.2.2 Slim binaries

The Slim Binary format [30, 21, 22] was originally developed as an extension of the modular Oberon system, in which this format was used to provide architecture-independent distribution of Oberon modules. The name Slim Binaries was chosen to contrast with that of Fat Binaries [46], a name used for commercial distribution formats from Apple and Next, which stored binaries for multiple program architectures in a single file. Since Fat Binaries store one version of the entire program executable for each machine architecture, Fat Binaries tend to be large and require a complex build process [28].

Slim Binaries avoid these disadvantages by using a portable and high-level intermediate representation, that is based on the encoded abstract syntax tree and symbol table of a program. In the extended Oberon system (see Figure 7), the producer-side translates Oberon modules into Slim Binary files and distributes them to several consumers. After successful transmission, a code-consumer can restore the syntax tree and symbol table from the r Slim Binaries and then verify its correctness. If this verification succeeds the syntax tree and symbol table are then used to generate the machine code of the target platform. In the actual implementation, a single code generating loader decodes Slim Binary files and generates code in an unified process.

The program representation contained in Slim Binary files consists of a compact description of the symbol table and a syntax-oriented encoding of the abstract syntax tree that is based on a technique called Semantic Dictionary Encoding (SDE).[5] In SDE the encoding is performed using a dynamically generated semantic dictionary table, in which each entry stands for a special type of node used in the abstract syntax tree. As a consequence, the abstract syntax tree of a program in Slim Binary format is not described through nodes directly, but through a sequence of indices, where each index stands for an entry in the dictionary table. The resulting sequence of indices is stored, conjoined with

[5]In principle, SDE is a clever application of the well-known LZW compression algorithm [71] on expressions.

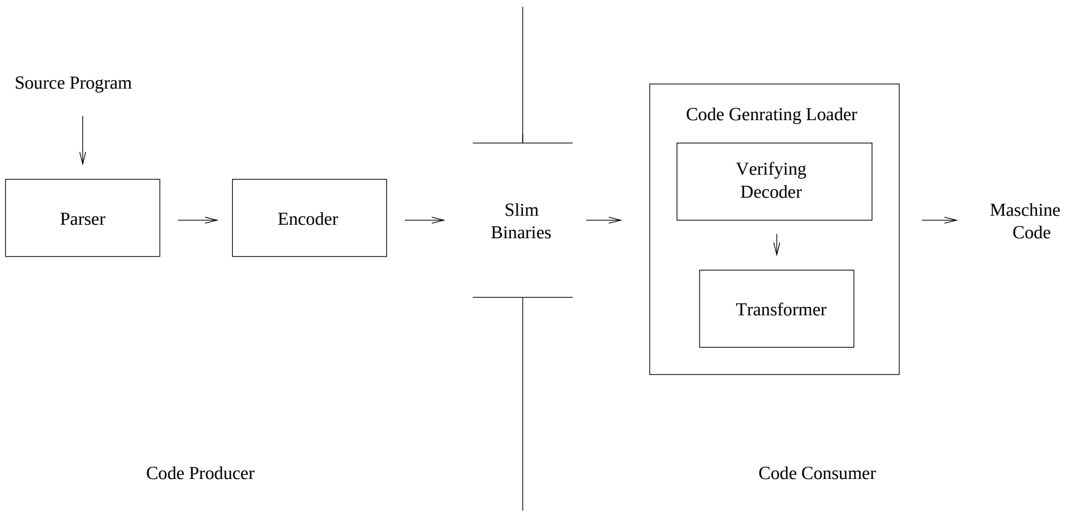


Figure 7: A Slim Binary Scenario.

the symbol table, in a file, which then can be transmitted to the code-consumer.

In SDE the dictionary table is generated in the exact same manner during encoding and decoding processes, therefore it is not necessary to store the dictionary table itself into a Slim Binary file. Instead, the dictionary table of a program is rebuild automatically during decoding of the abstract syntax tree on the consumer-side. Construction of the dictionary table is always performed in three steps. First, the dictionary table is filled up with entries that describe the control structures and operators (e.g., *if*, *while*, *for*, +, –, *, and /) of the used programming language. Second, the dictionary is augmented with entries from the symbol table for the variables and constants defined in the program. Finally, the dictionary can be enhanced with special entries, which we will describe in detail later.

Figure 8 (a) contains the abstract syntax tree and the corresponding symbol table for the same sample source program that was shown in Figure 6 (a), and Figure 8 (b) shows the sequence of indices resulting from applying the SDE. To simplify matters, the dictionary table shows only those entries which appear in the abstract syntax tree. In actuality, in order to describe all control structures and operators significantly more entries must be placed into the table. In SDE, a '.' stands for operands that have not yet been processed, e.g. if the entry '.=.' is selected, the left and right operands will need to be read. There are also dictionary entries (8 to 10 in our example) for each of the entries in the symbol table.

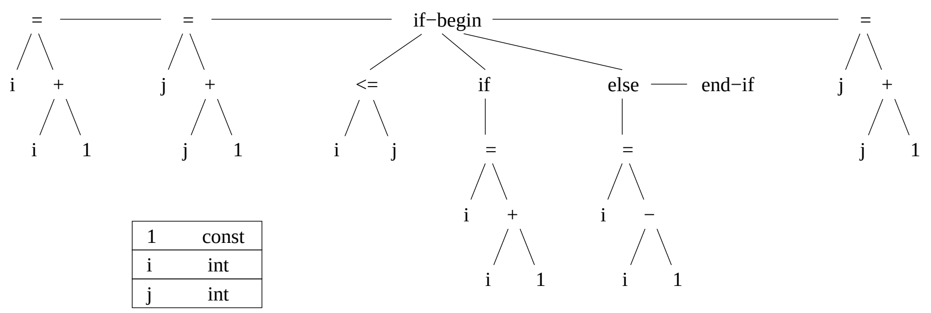
The dictionary table generated for our sample program can then be used for encoding the abstract syntax tree. For that purpose, the nodes of the abstract syntax tree are traversed in pre-order, and as each node is processed, the index of its corresponding node class is written out. For example, the expression *i=i+1* can be encoded as the sequence of indices, *4-9-5-9-8*, corresponding to this expression in prefix notation: *=i+i1*. Encoding of expressions in prefix notation allows the abstract syntax tree of a program to be rebuilt directly as the Slim Binary file is processed.

Application of this simple SDE encodes each assignment of the sample program using at most 6 indices. On closer inspection, it is apparent that certain sequences reappear multiple times within the Slim Binary file (e.g., the encoding of the first and third assignment are identical). The Slim Binary format allows for the compression of recurrences of similar patterns, by adding additional entries to the SDE during the encoding process that express patterns of nodes that have already been seen. As an example, after processing the assignment *i=i+1*, entries for the subexpressions *i=.*, *i+.*, *.+1*, *i+1*,*.=i+1* and *i=i+1* are inserted into the dictionary table. Figure 8 (c) shows excerpts of the dictionary table extension that would be adaptively built up during the encoding of our sample program. As can be seen, this SDE dynamic extension mechanism reduces the number of indices required for the sample program from 32 to 24 indices.

The insertion of additional entries for the description of these subexpressions increases the size of the dictionary table and with it the number of bits that are required for table index representation. However, in an optimized SDE, not all above discussed dictionary entries must be inserted into the dictionary table. Ref. [22] contains a detailed description of insertion strategies that can be used for effective construction of dictionary tables for the Slim Binary format. During decoding of an abstract syntax tree that has been encoded by a dynamic SDE, the same adaptive construction of the dictionary table must be performed. As a consequence, after the recovery of the expression, *i=i+1*, entries for subexpressions *i=.*, *i+.*, *.+1*, *i+1*,*.=i+1* and *i=i+1* must be inserted into the dictionary table in the same order on the consumer side as they were on the producer side, since otherwise the abstract syntax tree cannot be regenerated correctly.

The effectiveness of Slim Binaries as intermediate rep-

(a)



1	const
i	int
j	int

(b)

```
int i,j;
...
i = i + 1;        4   9   5   9   8
j = j + 1;        4  10   5  10   8
if (i <= j)       0   7   9  10
   i = i + 1;     1   4   9   5   9   8
else
   i = i − 1;     2   4   9   6   9   8
end if;           3
j = j + 1;        4  10   5  10   8
```

Index	Meaning
0	if−begin
1	if
2	else
3	end−if
4	.=.
5	.+.
6	.−.
7	.<=.
...	...
8	1
9	i
10	j
11	i=.
12	i+.
13	.+1
14	i+1
15	.=i+1
16	i=i+1
17	j=.
18	j+.
19	j+1
20	.=j+1
21	j=j+1
22	i<=.
23	.<=j
24	i<=j
25	i−.
26	.−1
27	i−1
28	.=i−1
29	i=i−1
...	...

(c)

```
int i,j;
...
i = i + 1;        4   9   5   9   8
j = j + 1;        4  10  13  10
if (i <= j)       0   7   9  10
   i = i + 1;     1  16
else
   i = i − 1;     2  11   6   9   8
end if;           3
j = j + 1;       21
```

Figure 8: A Slim Binary Example.

resentation for mobile code was demonstrated by the Juice browser plug-in [27], which allowed Oberon applets (compiled into Slim Binaries) to be executed locally inside the web browser (via Juice's code generator) just like Java applets. Since the Slim Binary format results in smaller file sizes than corresponding Java Bytecode files, transmission times of Juice applets are shorter than for equivalent Java applets [47]. Furthermore, many optimizations can be performed on Juice applets due to the retention of high-level program information.

Program optimizations that are performed on the consumer-side impose additional runtime costs. Therefore, instead of enforcing optimizations during load time, they can be performed as background process while the mobile code is already executed. The Slim Binary format is well suited for this kind of runtime optimization [48].

Within an environment that supports runtime optimizations, Slim Binaries are first transformed into the machine code of the target platform during load time without applying any program optimizations. Subsequently, while the machine code executes, additional transformations and program optimizations can be performed in a separate thread. With each transformation, the quality of the generated machine code is enhanced, until a certain level of optimization is achieved.

Runtime optimizations are also able to support complex transformations (e.g., inter-modular and approximative program optimizations). Extended variants (see, for example, [7, 62]) use adaptive analysis to identify frequently executed parts of the mobile code. Using this information, the optimizations can be performed more efficiently.

A variant of Slim Binaries for the Java language is implemented by the ASTCode format [66]. The main objective of this approach was to produce a more compact intermediate representation than Java Bytecode and to simplify the verification process on the consumer-side. In ASTCode the class file format has been changed slightly. In particular, the constant pool of the class files is used as a symbol table, and instead of Java Bytecode sequences, in ASTCode class files contain sequences of indices of the Semantic Dictionary Encoding. In order to simplify the verification process, the decoding process of a class file in ASTCode is extended by a type-checking procedure. As a result, the complexity of the verification process, which is quadratic for Java Bytecode, is reduced to a linear function of code length.

3.2.3 SafeTSA

We also classify SafeTSA (which stands for Safe Typed Static Single Assignment Form) as a tree-oriented intermediate representation for mobile code, even though it is actually a hybrid format that combines high-level control structures in a AST-like form (called the Control Structure Tree) with individual instructions in static single assignment form [3, 70]. The format was designed as a drop-in replacement for Java Bytecode[6] providing for more efficient just-in-time compilation and an innovative approach to safety based on an *inherently safe* encoding.

SafeTSA's control structure tree provides for all of the non-linear intra-procedural control flow in SafeTSA. The instructions (which only perform computations, manipulate data on the heap, and call methods) are embedded as leaves of the control structure tree with their execution being controlled by their parents in the tree. The high-level control structures provided by SafeTSA (which mirror those provided by the Java programming language), restrict SafeTSA programs to reducible intra-procedural control flow. They also make it possible to do a syntax directed derivation of the control flow graph and dominator tree, and also allow for the possibility of high-speed single-pass syntax-directed JIT compilation of SafeTSA code.

The primary driver of enhanced efficiency for just-in-time compilation of SafeTSA, however, results from the use of Static Single Assignment Form (SSA). Static Single Assignment Form guarantees that each instruction's result variable is unique (i.e., assigned to at only that static location in the program) [14]; this discipline (which is facilitated by special ϕ-functions that merge alternative values that reach a program point on different control flow paths) enables a variety of optimizations that are now standard in state-of-the-art optimizing compilers. In SafeTSA, the use of SSA facilitates producer-side machine-independent optimization and speeds up several consumer-side optimizations. As reported in [4], the net result is that JIT compilers for SafeTSA can deliver the same quality code in less time than a JIT compiler for JVML.

Static single assignment form also plays a key part in SafeTSA's inherently safe encoding. The binary on-the-wire SafeTSA is designed such that it only uses the number of bits required to represent possible program symbols that might result in a syntactically valid and correctly typed program [70]. In this way, the program is more dense, because it is not wasting bits that do not differentiate between correctly typed programs.

In addition, a separate verification phase is unnecessary, because the decoding process only ever produces syntactically valid and correctly typed programs. There are a couple of mechanisms that enable this. Perhaps the most important mechanism is the implicit naming and enumeration of variables according to *dominator scoping* and the *type separation*. The implicit naming is based on the property that, in static single assignment form, each variable is only ever assigned at a single location, so by enumerating the locations where variables are created, one can create names for the variables. In static single assignment form, a variable is live at a program point, if and only if, its defining instruction dominates that program point. Therefore, SafeTSA limits the scope of all SSA variables to the program region dominated by its definitions, and the implicit enumeration takes advantage of this so that variables are enumerated consecutively along the path of the dominator tree to the point the variable is being accessed.

In addition SafeTSA's variable enumeration is *type separated*. That is, there are no implicit coercions, so the variables of each type can be enumerated independently. These mechanisms enable all symbols representing operands to be selected from a list of candidate operands that would be legal in that program location. Simpler mechanisms are used for symbols and other kinds of program elements, and a binary prefix code is generated for each position in the program based on an implicit enumeration of the possible alternative symbols for that position.

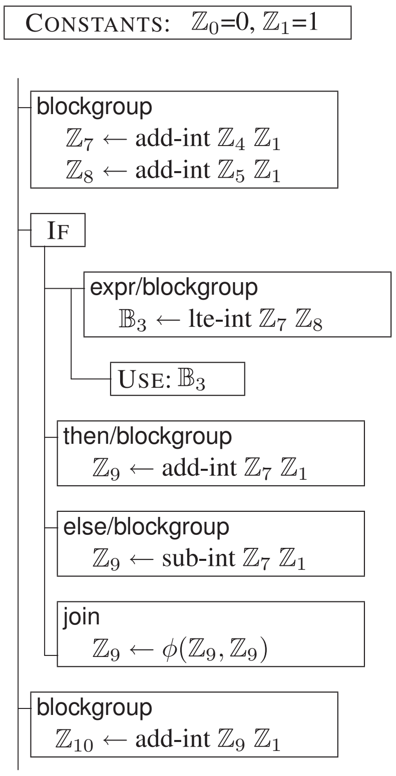
All of these mechanisms can be seen in Figure 9. The section of the control structure tree shown in Figure 9(b) contains a node for the IF statement, a node naming the boolean value that should be "used" to control the IF statement, and several blockgroups, which contain instructions, and some of which are subordinate to the IF statement in particular ways (e.g., the THEN-statement). In order to make the code easier to read, the variables are named with a

[6]And, in fact, the prototype implementation of SafeTSA based on the Jikes Research Virtual Machine supports intermixing classes loaded from both SafeTSA and JVML class files within a single executing virtual machine [4].

```
int i,j;

i = i + 1;
j = j + 1;
if (i <= j)
    i = i + 1;
else
    i = i − 1;
j = j + 1;
```

(a) in Java


CONSTANTS: $\mathbb{Z}_0$=0, $\mathbb{Z}_1$=1
blockgroup
$\mathbb{Z}_7 \leftarrow$ add-int $\mathbb{Z}_4$ $\mathbb{Z}_1$
$\mathbb{Z}_8 \leftarrow$ add-int $\mathbb{Z}_5$ $\mathbb{Z}_1$
IF
expr/blockgroup
$\mathbb{B}_3 \leftarrow$ lte-int $\mathbb{Z}_7$ $\mathbb{Z}_8$
USE: $\mathbb{B}_3$
then/blockgroup
$\mathbb{Z}_9 \leftarrow$ add-int $\mathbb{Z}_7$ $\mathbb{Z}_1$
else/blockgroup
$\mathbb{Z}_9 \leftarrow$ sub-int $\mathbb{Z}_7$ $\mathbb{Z}_1$
join
$\mathbb{Z}_9 \leftarrow \phi(\mathbb{Z}_9, \mathbb{Z}_9)$
blockgroup
$\mathbb{Z}_{10} \leftarrow$ add-int $\mathbb{Z}_9$ $\mathbb{Z}_1$

(b) in abstract SafeTSA

Symbol	Index/Choices	Encoding
statement blockgroup	1/12	001
apply	19/20	11111
add-int	89/185	10100000
$\mathbb{Z}_4$	4/7	101
$\mathbb{Z}_1$	1/7	010
add-int	89/185	10100000
$\mathbb{Z}_5$	5/8	101
$\mathbb{Z}_1$	1/8	001
end blockgroup	0/20	0000
IF	3/12	011
expression blockgroup	1/3	10
apply	19/20	11111
lte-int	104/185	10101111
$\mathbb{Z}_7$	7/9	1110
$\mathbb{Z}_8$	8/9	1111
end blockgroup	0/20	0000
use:	—	—
$\mathbb{B}_3$	3/4	11
then:	—	—
apply	19/20	11111
add-int	89/185	10100000
$\mathbb{Z}_7$	7/9	1110
$\mathbb{Z}_1$	1/9	0001
end blockgroup	0/20	0000
else:	—	—
apply	19/20	11111
sub-int	111/185	10110110
$\mathbb{Z}_7$	7/9	1110
$\mathbb{Z}_1$	1/9	0001
end blockgroup	0/20	0000
join:	—	—
ϕ	0/2	0
$\mathbb{Z}_9$	9/10	1111
$\mathbb{Z}_9$	9/10	1111
end join	1/2	1
statement blockgroup	1/12	001
apply	19/20	11111
add-int	89/185	10100000
$\mathbb{Z}_8$	8/11	1101
$\mathbb{Z}_1$	1/11	001

(c) in SafeTSA's Binary Encoding

Figure 9: The Example Program Fragment in SafeTSA

symbol representing the type ($\mathbb{Z}$ for integer, $\mathbb{B}$ for boolean) and a subscript indicating the variables position in 0-based implicit enumerations. The integer constants, 0 and 1, are declared to be represented by $\mathbb{Z}_0$ and $\mathbb{Z}_1$, respectively. The initial values of i and j are assumed to be $\mathbb{Z}_4$ and $\mathbb{Z}_5$, and it is assumed that there are 7 integers and 2 booleans defined before the first instruction shown. With these assumptions, the first instruction in the first blockgroup adds 1 to $\mathbb{Z}_4$ (i.e., the old i) and puts the result in $\mathbb{Z}_7$ (i.e., the new i). Note that there are several definitions of $\mathbb{Z}_9$, which appears to be a violation of the single assignment property, but none of these definitions dominates any of the others so their scopes do not overlap and they are distinct variables. In fact, this mechanism effectively prohibits accessing non-dominating variables, since their names get re-used by those that do dominate a particular access. Due to the peculiarities of ϕ-functions in SSA, the definition of the third $\mathbb{Z}_9$ actually refers to the first $\mathbb{Z}_9$ and the second operand refers to the second $\mathbb{Z}_9$, but according to SafeTSA's rules [70], only the correct $\mathbb{Z}_9$ is in scope at each of those positions. The rendering of the tree representation into a sequence of symbols, and the binary encoding of those symbols is shown in Figure 9(c).

3.3 Proof-annotated representations

In the past decade, there have been several research projects aiming at the development of certifying compilers. Certifying compilers differ from traditional compilers in that in addition to producing executable code, they also produce an additional annotation (i.e., a certificate) containing a proof that the executable code respects certain safety properties (usually type and memory safety). The proof-annotated code format is designed so that all proofs can be automatically checked in a bounded amount of time. In a mobile code context, such a proof-annotated format can be used to only allow the execution of mobile code for which it is determined that the proofs are correct and that the proofs are sufficient to guarantee that the annotated code satisfies the safety properties required by the mobile code system.

Proof-Carrying Code [55] and Typed Assembly Language [51] are the two primary representatives of proof-annotated mobile code formats. As introduced by George Necula in 1996, proof-carrying code utilizes certificates written in a formalism based on first-order logic. This proof can be generated by the code producer and shipped along with the program code. The code consumer then validates the proof to ascertain the safety of the transmitted mobile code. Due to its foundation in first-order logic, proof-carrying code is quite flexible in terms of the types of safety properties that can be checked using first-order logic; the limiting factors on flexibility are the kinds of properties for which proofs can be generated automatically. The Touchstone compiler[7] is the front-end of a prototype proof-carrying code system that compiles from a safe subset of the C programming language into machine code and certifies that the resulting machine code is type and memory safe [55, 58].

Typed Assembly Language extends traditional untyped assembly languages with typing annotations, memory management primitives, and a sound set of typing rules. These typing rules guarantee the memory safety, control flow safety, and type safety of the transmitted program.

3.3.1 Proof-carrying code

Proof-Carrying Code (PCC) was originally developed as a mechanism for safe operating system kernel extensions, but was later adapted to the area of mobile code [54, 56]. Founded on formal program verification theory, PCC allows the code consumer to check the safety of programs by checking machine-readable proofs that are generated by the code producer and shipped along with the program code. After checking the validity of the proofs, code consumers are then assured that the program execution will not violate the verified properties. The desired safety properties depend on the code consumers safety policy, which acts as a contract between the code producer and code consumer, and defines which conditions must be satisfied by safe mobile programs.

In a proof-carrying code system, the role of the code producer is fulfilled by a certifying compiler, consisting of an annotating compiler and a proof-generator. While the certifying compiler translates the program that is to be transmitted into machine code of the target platform or any other executable code representation, it also annotates the program with additional information (e.g., types) that would otherwise be lost. After this, a theorem (possibly-specialized) prover is used to generate a proof that the generated code complies with the mobile code system's safety policy, and this proof is transmitted along with the code to the code consumer in a PCC binary. The code consumer validates the safety proof based on the actual machine code and the conditions defined in the safety policy. The validation algorithm and the safety policy are the only parts of the system which have to be trusted, minimizing the size of the trusted code base (TCB)[8]. If proof validation is successful, the safety of the transferred mobile program is guaranteed and the machine code can be executed as shown in Figure 10.

Safety conditions, which have to be satisfied by the transferred mobile code are defined within the safety policy and are shared between code producer and code consumer. This safety policy is based on first-order logic and consists of three parts: a verification condition generator (VCG) [19], a set of axioms, and pre- and post-conditions.

The verification condition generator is used to derive a safety predicate (i.e., a verification condition), from the annotated program code. The safety predicate is derived such that it will only evaluate to true if every condition specified

[7]A variant of Touchstone, called SpecialJ, has been developed for the Java programming language [10, 11].

[8]The paper [5] describes an even further reduced trusted code base which incorporates the safety policy into the proof.

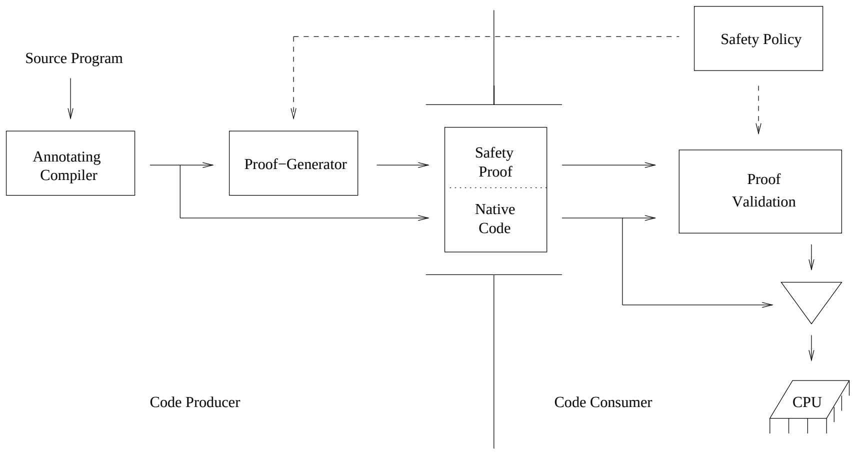


Figure 10: Proof-Carrying Code Architecture.

in the safety policy is satisfied. The pre- and post-condition included in the safety predicate, express constraints on the machine state that must hold, respectively, before and after program execution. The safety predicates, which are defined in first-order logic, are derived from a set of axioms and derivation rules that model the state transitions of the target machine associated with each instruction.

Both, code producer and code consumer, derive the safety predicate from the annotated program code. The code producer generates a proof of the safety predicate, indicating the safe execution of the program code, and the code consumer derives the safety predicate in order to check the matching of program code and safety proof. Thus, the verification condition generator traverses the program code and creates predicates for each critical instruction (e.g., memory access) using a symbolic interpreter, such that a proof of these predicates ensures that executing the corresponding instruction does not invalidate the conditions defined in the safety policy.

To support complex program structures like method calls and loops, the verification condition generator uses *invariants* annotated in the program code. These annotations are frequently required to mark loop invariants, which cannot normally be automatically derived by the verification condition generator. The invariants are included among the predicates which must be verified. Thus, the code consumer does not need to trust the program annotations, and the invariants are only used as hints supporting the generation, and the validation of the safety proof. These predicates must be proved to hold for every control path between two distinct invariants, starting with the pre-condition and finishing with the post-condition. As a consequence, the safety predicate of the whole program is the conjunction of all predicates derived from the invariants and the individual instructions.

After the safety predicate has been derived by the verification condition generator, the code producer creates a proof, which shows the correctness of the generated safety predicate. This safety proof is represented using the Edinburgh Logical Framework or LF notation [42, 68]. The Edinburgh Logical Framework efficiently validates the proofs by reducing validation to a simple type-checking procedure [55, 57]. (In other words, in the Edinburgh Logical Framework, only correct proofs are correctly typed.) Thus, the code consumer of a PCC system can be guaranteed that a mobile code program satisfies its safety policy prior to its execution.

As a concrete example of a PCC system, let us examine the output of the Touchstone certifying compiler. The Touchstone certifying compiler might translate the source program, shown in Figure 11 (a), into the slightly optimized DEC ALPHA assembly code as shown in Figure 11 (b). Note that a pre- and post-condition are annotated, both stating that registers v0 and t0, which represent variables i, j of the source program respectively, contain integer values. In order to verify the safety of the machine code, the Touchstone certifying compiler generates a verification condition and a safety proof, indicating the validity of the verification condition. The verification condition, shown in Figure 11 (c), states type and memory safety of the machine code. Therefore, the verification condition denotes the implication that: assuming the registers v0 and t0 hold integer values initially, their values after manipulation will still be of type integer after the machine code has been executed. Thus, the post-condition can be derived from the pre-condition, and the validity of the verification condition is proven. The safety proof shown in Figure 11 (d) states the validity of the verification condition and therefore guarantees type and memory safety of the machine code.

One drawback of PCC is the size of the generated safety proofs, especially since the proofs have to be transmitted along with the code to the code consumer. The transfer of the safety proof conjoined with the program code is performed using a PCC binary, which consists of three parts. First, the program to be transferred is included using an intermediate representation or the machine code of the target platform. In the latter case, the program can be directly loaded and executed after the mobile code has been suc-

```
(a) source program

    int i, j;
        i = i + 1;
        j = j + 1;
        if (i <= j)
                i = i + 1;
        else
                i = i - 1;
        j = j + 1;

(b) annotated machine code

        ANN_PRE(example, LF_
                        (/\ (of t0 int)
                            (of v0 int))_LF)

        subl    t0, v0,t1
        blt     t1, L2
        lda     v0, 2(v0)
    L2:
        lda     t0, 2(t0)

        ANN_POST(example, LF_
                        (/\ (of t0 int)
                            (of v0 int))_LF)

(c) verification condition

pf (all ([X0 : exp]
          (all ([X1 : exp]
               (=> (/\ (of X1 int) (of X0 int))
                   (/\ (=> (>= (- X1 X0) 0)
                           (/\ (of (+ X1 2) int)
                               (of (+ X0 2) int)))
                       (=> (< (- X1 X0) 0)
                           (/\ (of (+ X1 2) int)
                               (of X0 int))))))))

(d) proof

(alli [X0 : exp]
 (alli [X1 : exp]
  (impi [A1 : pf (/\ (of X1 int) (of X0 int))]
   (andi (impi [A2 : pf (>= (- X1 X0) 0)]
     (andi (ofIntAny (+ X1 2))
      (ofIntAny (+ X0 2))))
    (impi [A3 : pf (< (- X1 X0) 0)]
     (andi (ofIntAny (+ X1 2))
      (ander A1)))))))
```

Figure 11: A sample program (a) and its PCC output (b), (c) and (d).

cessfully verified. The second part of the Proof-Carrying Code binary contains a symbol-table, which is used to reconstruct the LF representation of the safety proof on the consumer side. The last part includes the safety proof in a binary encoding.

3.3.2 Typed assembly language

The use of a Typed Assembly Language as intermediate representation benefits a mobile code system in several ways. First of all, a number of program optimizations are enhanced by having type information available in the assembly code. In addition, the type annotations facilitate the verification of the mobile code's type safety. In order to gain these benefits, a type abstraction of assembly languages is required, which guarantees type safety of well-formed assembly programs and hence enables the transformation of well-formed input programs into safe assembly code [38]. Since the type annotations and type checking, serves to prove that a program in a Typed Assembly Language is type safe, it can be considered as a kind of proof-annotated representation, even though the proof is not that of a direct assertion of safety in a general logic as it is in proof-carrying code [52].

The feasibility of Typed Assembly Language was demonstrated by the reference implementation, TALx86

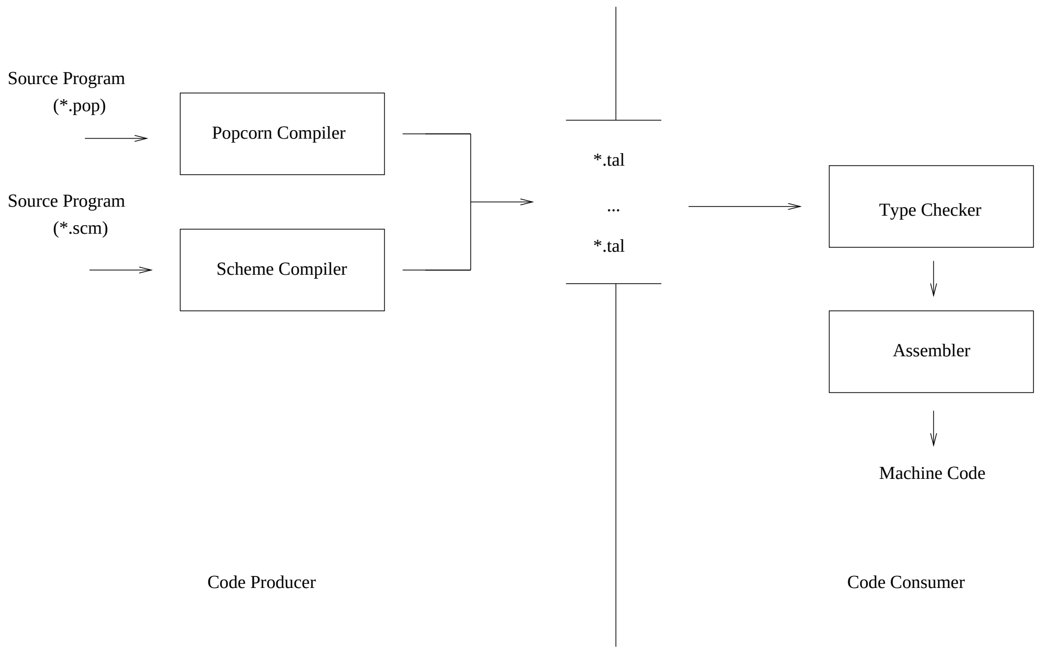


Figure 12: Talx86 system.

[36]. TALx86 incorporates support of the programming languages Scheme and Popcorn (a type-safe subset of the C language from which unsafe constructs such as pointer arithmetic and address operator have been removed). On the producer side of the prototype TALx86 system (see Figure 12), programs of the Scheme or Popcorn programming language are transformed into assembly code for the target platform, and the generated assembly code is annotated with type information resulting in Typed Assembly Language. After receiving a program, the code consumer uses the annotated type information to type-check the transferred assembly program. If the mobile code is successfully verified, an assembler transforms the assembly code into machine code for the target platform, which is then executed.

The TALx86 implementation is based on Microsoft's Macro Assembler Language, and therefore, TALx86 programs, after being type-checked, can be efficiently assembled with common commercial assemblers. Within the TALx86 system, the register-based Microsoft Macro Assembler Language is extended with annotations, which are mainly used as pre-conditions of code labels, assigning type information to registers.

Our sample program (now written in Popcorn) in Figure 13 (a) and its translation into TALx86 are shown in Figure 13 (b). The TALx86 sequence consists of two sections: the assembly language instructions and its corresponding type annotations. The assembler program starts by calculating increments of values contained in registers EBP and EDI, which represent the variables i and j, respectively. Subsequently, the values are compared, and if the value contained in EBP exceeds the value in EDI, execution continues at the code label ifFalse$49, which represents the else branch of the program. Otherwise, the then branch of the if statement is executed, and the value contained in EBP is incremented. After that, program execution proceeds to the label ifMerge$50, which marks the end of the if statement, and the value in register EDI is incremented again.

Both labels are annotated with the types of values contained in the registers at that point. As an example, annotation EDI:B4 denotes type B4 in register EDI, indicating a 4-byte integer value inside. On the consumer side, the annotation is then used by the type-checker, so that it only has to check that the register EDI contains a value of type B4 before control is given to label ifFalse$49 or ifMerge$50 (rather than propagating types around the control-flow graph until a fixed pointer is reached or a type error is detected).

Polymorphic types, required for describing high-order structures like stacks, are realized using placeholders, which are replaced by corresponding types before control is transferred to the associated code label. Type annotations are also used to define new types with type constructor declarations. This flexibility of the type system allows one to support a wide range of programming languages. TALx86 allows the code consumer to provide routines of instructions (i.e., macros) for manipulating complex data structures that can be typed and treated as atomic operations during verification. Programs may explicitly allocate such complex data structures using the macro malloc but are not allowed to explicitly de-allocate the structures; this is done implicitly, through garbage collection.

Stacks in TALx86 programs are modeled by abstractions that are based on lists. The expression t :: s denotes a stack,

```
(a)     int i, j;
        i = i + 1;
        j = j + 1;
        if (i <= j)
                i = i + 1;
        else
                i = i - 1;
        j = j + 1;

(b)     MOV     EDX,EBP
        ADD     EDX,1
        MOV     EBP,EDX
        MOV     ESI,EDI
        ADD     ESI,1
        MOV     EDI,ESI
        CMP     EDX,ESI
        JG      tapp(ifFalse$49,<r$9>)
        ADD     EBP,1
        JMP     tapp(ifMerge$50,<r$9>)
ifFalse$49:
LABELTYPE <All[r$9:Ts].{EDI: B4,EBP: B4,ESP: sptr {ESP: sptr r$9}::r$9}>
        MOV     ESI,EBP
        SUB     ESI,1
        MOV     EBP,ESI
        FALLTHRU        <r$9>
ifMerge$50:
LABELTYPE <All[r$9:Ts].{EDI: B4,EBP: B4,ESP: sptr {ESP: sptr r$9}::r$9}>
        ADD     EDI,1
```

Figure 13: Sample program in Popcorn (a) and corresponding TALx86 program (b).

consisting of a top-most element of type t and the rest of the stack described by s. Placeholders are applied in order to enable the polymorphic representation of stacks. A placeholder is of a general form All[s:Ts] where Ts denotes the abstract type, which is substituted by a corresponding instance s.

As a consequence, function calls are represented by the help of a runtime stack s, which is referenced by stack pointer sptr s contained in register ESP. This can bee seen at labels ifFalse$49 and ifMerge$50 of Figure 13 (b), where register ESP references the runtime stack, which contains another stack pointer representing the caller frame, and the rest of the stack denoted by the polymorphic type r$9.

Polymorphic types in combination with runtime stacks are also used to implement visibility rules, which make the actual representation of abstract types associated to local variables only resolvable by the authorized function. Furthermore, this mechanism supports exception handling by restricting register access. A dedicated register contains a stack pointer which indicates where to unwind the runtime stack to. This stack pointer is typed so that it is abstract and therefore unmodifiable by everything except for the exception code.

The time-critical processes in the TAL system include the code consumer's type-checking and the transfer of the mobile program to the code consumer. Thus, a compact encoding of Typed Assembly Language is needed for optimal performance. Since the annotations (e.g., pre-conditions of code-labels) increase the code size, various compression techniques can be applied to increase the density of the annotation format, so that it is more suitable for mobile code [37]. These techniques include, among other, the sharing of common sub-terms within annotations, the use of generic type abbreviations, and the elimination of unnecessary annotations.

4 Review and comparison

In the following sections, the mobile code representations that were presented earlier will be evaluated according to the requirements introduced in Section 2. Table 1 summarizes this evaluation and can be used as a guide to the discussion that follows.

4.1 Source language flexibility

Although the JVM was designed to support Java semantics, it can also be used as a target for other languages. Indeed, several compilers for C, C++, and Ada95, target the JVM. However, these language's intrinsic insecurities, and their semantic mismatch with Java, require the programmer to adhere to restrictive feature subsets [32]. In order to avoid such disadvantages, .NET and its intermediate representation CIL were designed to efficiently support a variety of object-oriented, functional, and procedural programming languages, including C#, C++, Java, Fortran, Cobol, Eiffel, Haskell, ML. Furthermore, the .NET platform's Common Type System serves as a common denominator that aids cross-language interoperability, so that .NET components can interact with each other even if they are written

	Stack-based		Tree-oriented			Proof-Annotated	
	JVML	CIL	ANDF	Slim Binary	SafeTSA	PCC	TAL
Flexibility							
Input-Languages	✓	+	✓	✓	✓	++	+
General type system	−	++	−	−	−	−	++
Portability							
Target-Architectures	++	++	+	+	+	−	−
Compactness							
Encoding Density	✓	✓	+	++	+	−	−
Efficiency							
Interpreter	++	++	✓	✓	✓	−	−
JIT compiler	✓	✓	+	+	++	✓	✓
Producer Optimizations	✓	✓	+	+	+	++	++
Producer Annotations	✓	✓	✓	✓	++	++	++
Consumer Optimizations	+	+	+	+	+	✓	✓
JIT Cost	✓	✓	+	+	++	✓	✓
Safety							
Safety	++	++	−	++	++	++	++
Automated	++	++	−	++	++	−	++
Runtime Complexity	✓	✓	N/A	+	+	+	+

Legend: −poor/no, ✓adequate, +good/yes, ++excellent.

Table 1: Intermediate representations in comparison.

in different CTS-supporting languages.

Tree-oriented intermediate representations tend to be more limited in their linguistic flexibility. The current ANDF-System supports Ada95 and C; Slim Binaries' and SafeTSA's prototype systems are built to support the source language Oberon and Java, respectively. Although, it seems that tree-oriented techniques are limited to programs written in predetermined languages, representatives of this kind of intermediate representation also can be extended for addressing a multiplicity of source languages. Basically, such an extension might be based on the construction of an unified abstract syntax tree and a more general type system.

In principle, the greatest source-language flexibility can be achieved with proof-annotating intermediate representations, since for most programming languages, a front end, which translates a source program into native code, can be easily constructed. Furthermore, a principle objective of the TAL project was the development of a statically typed, low-level intermediate representation, that could be used for multiple source languages and on which multiple program optimizations could be performed [64]. For the description of type systems of different source languages, the TAL system transforms a program internally into an intermediate representation that is based on a high-order λ calculus, from which eventually after several type-conserving restructurings the TAL program is derived.

4.2 Portability

A code consumer can execute mobile code applications using an interpreter, a JIT compiler or both. On most current desktop and server computer systems adaptive JIT compilation techniques provide the best performance. However, as small resource constrained devices (e.g., cell phones, PDA's, Java cards) become more and more ubiquitous, interpreters in mobile code systems have become more important, since compared to compilation, interpretation usually uses less resources.

Stack-oriented intermediate representations provide an excellent foundation for the development of fast and efficient interpreters. In contrast, interpretative program execution is supported from none of the tree-oriented prototype systems, as these mobile code formats mainly focus on JIT compilation. Nevertheless, tree-oriented systems can include interpreters (see e.g. [41, 31]), which may be slower than for stack-based counterparts, but could be used for program execution on platforms for which JIT compilers are not yet available.

JIT compilers that transform JVML and CIL programs, respectively, into machine code have been developed for the most common computer systems. Although all of the presented tree-oriented systems are developed for a restricted number of architectures, in principle, these intermediate representations can be considered just as portable. That is, since targeting other architectures needs only more engineering resources to implement new back-ends translating tree-based program representations into their corresponding machine code.

Although PCC and TAL, in their original incarnations, are based on the target machine assembly language, and thus are not portable. In principle, they could also be used as input for JIT compilation, in which case, the assembly language could be replaced with a more general register-based language. A candidate for such an all-purpose low-level language could be the intermediate representation used by the VCODE system (which is the machine code of an idealized RISC-Architecture) [18]. This would serve as a common target language for programs written in different programming languages and as input for an on-the-fly machine code generation of different architectures.

4.3 Compactness

Compactness of mobile code applications plays a major role, especially as many today's network connections are wireless and have a limited bandwidth. In such networks, raw throughput rather than network latency is the main bottleneck. Moreover, increasing use of mobile code on constrained devices, also puts attention on the size of program representation due to limited memory resources.

The use of tree-oriented intermediate representations usually leads to better file sizes than stack-based techniques. In particular, according to measurements described in [24], Slim Binaries are more dense than compressed JVML class files by a factor of 1.72. And for uncompressed JVML the ratio in file sizes in average even can increase up to 2.42. SafeTSA, which has a hybrid tree/SSA structure, is not quite as dense, but as reported in [70], has a binary on-the-wire file size similar to compressed JVML class files.

Stack-based intermediate representations, in turn, are often more compact than the corresponding machine code [24]. Proof-annotating intermediate representations are still larger, because in addition to be based on less compact machine code or assembly language, the file sizes of proof-annotating intermediate representations are also increased by their proof or type annotations. Unfortunately, there exists no measurements about file sizes of proof-annotated code compared to JVML programs. However, measurements in a prototype PCC system resulted in an average ratio of proof size to code size of 2.5 [58]. Comparable experiments performed in the TAL system led to a ratio of up to 0.67 [37]. These results indicate a significant increase in file sizes when applying proof-annotating techniques, and consequently a need of sophisticated compression techniques.[9]

4.4 Efficiency

We call the property of an intermediate representation to support a fast and resource-efficient program execution, the representation's efficiency. Although it is a matter of common knowledge, that fast program execution is primarily achieved through the use of JIT compilation techniques, the efficiency with which a mobile code format can be interpreted is also important, especially as resource-constrained devices become more ubiquitous.

Stack-oriented intermediate representations are excellent candidates for interpretation. The main advantage of this architecture as input for an interpreter, is the compact instruction encoding (due to most operands being taken off the stack). Although, tree-oriented mobile code formats can also be interpreted, tree-based interpreters are not such efficient than its bytecode counterparts, because of a higher storage consumption and slower execution times, which are direct consequences of its internal representation as pointer structures. Register-based interpreters are also possible [65, 15], but have not been employed in industrial strength mobile code systems.

In recent years a lot of powerful JIT compilers for stack-based mobile code formats have been developed; especially notable are Sun's HotSpot compiler [62, 6] and IBM's Jikes RVM [7, 44], respectively. However, the popularity of existing stack-based JIT compilers belies the limitations of stack-based intermediate representation when used as input for a JIT compiler. Certainly, simple machine code for stack-based programs can be generated quickly, but for aggressive JIT compilation (i.e., with several complex optimizations) stack-based representations have some disadvantages.

The main disadvantage for aggressive JIT compilation of stack-based code is the use of the stack model. This approach requires the compiler to generate optimized register-based machine code for a program that is expressed in terms of the manipulation of a virtual stack machine. Most existing stack-based JIT compilers solve this problem by expending compilation effort to transform their input programs into an internal three-address code representation (often in SSA form) on which the optimizations are performed.

A further disadvantage is the low-level character of stack-based program code, which often prevents reconstruction of high-level language information, which is essential for certain optimizations. In addition, performing machine independent optimization on the producer side of a stack-based system is difficult. For example, while a compiler generating stack-based JVML code could, in principle, perform common subexpression elimination and store the resulting expressions in additional, compiler-created local variables, this approach introduces additional instructions and temporary variables that may negate any improvements created by the common subexpression elimination.

In contrast, due to its high-level entities, tree-oriented code formats are excellent candidates for JIT compilation. In principle, JIT compilers based on these intermediate representations can be just as effective as static compilers. The main advantage of a tree-oriented JIT compilation is the preservation of high-level information that aides the quick generation of fast code (e.g., explicitly marked loops, loop invariant codes, exclusion of irreducible control flow graphs).

SafeTSA successfully augments a tree-oriented intermediate representation with instructions in SSA form, which is already used internally in several static and JIT compilers and that is considered the state-of-the-art intermediate rep-

[9]as described for TAL in [37] and PCC in [59]

resentation for intra-procedural scalar optimizations. Several efficient optimization techniques have been developed for SSA programs in the last decade. Experiments, described in [4], confirm that JIT compilers using SafeTSA run faster than those using JVML code, reducing the cost for dynamic optimization of some programs by up to 90%. As mentioned above, the machine-independent optimization of stack-based mobile code formats is often awkward, but for tree-oriented and proof-annotated formats, such optimizations cause no further difficulties.

In recent years program annotations have been suggested as a way to improve the code generation of JIT compilers. The term program annotation is used as a synonym for code information added to the mobile code during its generation. This information can be used by the consumer side of a mobile system to speed-up optimizations of a given program. In principle, all types of intermediate representations support the transport of program annotations. The main challenge after transferring mobile code to the runtime environment is the verification of the transmitted annotations. Conceptually, verifiable program annotations can be constructed for PCC and TAL programs through proof and type extensions, respectively. In SafeTSA programs the concept of type separation can be applied in a tamper-proof manner for the safe transport of program annotations [43].

4.5 Safety

Safety is an important criterion in a mobile code system due to the inherent separation of code consumers and code producers. In general, mobile code can be created by an untrusted code producer and transferred through insecure communication channels to the code consumer, so the code consumer needs to verify that the transmitted mobile code will not perform any unsafe actions when executed.

In addition to other mechanisms such as cryptographic signatures, intermediate representations of mobile code address the safety issue using several distinct approaches, ranging from implicitly legal program encodings to formal methods like program verification using first-order logic as applied by Proof-Carrying Code. Common to all of them is the focus on guaranteeing type and memory safety as well as a legal control flow of the verified mobile program in order to provide fine grain isolation of code within the execution environment.

Stack-based intermediate representations for mobile code utilize a data-flow analysis in the verification process. This data-flow analysis is required due to the semantic gap between high-level source language and low-level intermediate representation [66]. Hence, as in the case of Java Bytecode, well-formed Java Bytecode sequences do not necessary represent legal Java programs. Adherence to certain safety concepts of the high-level source language is therefore verified using the data-flow analysis, which is performed on the consumer side of the mobile code system and targeted at type and memory safety as well as a legal control flow of inspected bytecode.

In addition to some semantic errors in original specifications and implementations of data-flow analysis for Java Bytecode verification [67], this approach also suffers from its immense costs, requiring quadratic time regarding the number of verified instructions in the worst case [66]. Because all of the verification work has to be done by the code consumer, this factor introduces another point of attack. Furthermore, since Java Bytecode verification assumes the type system and other safety concepts of the Java programming language, extending the underlying data-flow analysis to other programming languages and safety concepts is complicated. Due to its support for a wide range of programming languages, the Common Intermediate Language is more flexible on this point.

Two of the tree-oriented intermediate representations for mobile code, Slim Binaries and SafeTSA, represent one approach to avoid the semantic gap between source language and intermediate representation, and consequently facilitate the verification process. ANDF, the third tree-oriented intermediate representation, does not integrate a verification mechanism and so will not be discussed further in this respect.

The Slim Binaries format, as well as SafeTSA, implements a program encoding which is based on the abstract syntax tree and hence close to the high-level source language. Furthermore, both formats restrict their expressiveness to legal programs, (i.e., code violating safety criteria of the source language like type or memory safety can not be encoded by a well-formed Slim Binaries or SafeTSA program [25]. Thus, verification of mobile code is essentially done by checking the adherence to the general format of the corresponding intermediate representation.

As a consequence, complexity of the verification process can be reduced to linear time with regard to code length, as in the case of ASTCode [66]. The Slim Binaries format and its variant ASTCode have been designed for the Oberon and Java programming languages, respectively, hence addressing other languages with differing safety properties is a non-trivial task. This drawback also relates to SafeTSA, though it provides a mechanism for safe program annotations, which may be utilized in a broadened verification process incorporating extended safety criteria [43].

Proof-Carrying Code is based on the concept of certifying compilers (i.e., that produce a machine-readable safety certificate to accompany the mobile code and guarantee its safe execution). Due to the formal representation of the certificate using first-order logic and the small trusted code base [58, 5], Proof-Carrying Code can be seen as an intrinsically safe intermediate representation for mobile code.

Furthermore, the genericity of the underlying approach allows the incorporation of extended safety criteria by adapting the logic of the safety policy and the proof generator. Current applications of Proof-Carrying Code, however, are typically limited to type and memory safety. The main drawback of Proof-Carrying Code also relates to its foundation on formal program verification theory: loop invariants must be annotated as part of the safety proof generation, but these invariants cannot always be automatically inferred from the program, so manual annotations may be necessary if the properties to be proved are more complex than type safety in a tractable type system.

Because creating the safety certificate is expensive, proof generation has been shifted to the producer side of the corresponding mobile code system. The code consumer needs only to verify the shipped proof using a type-checking procedure, requiring linear time with regard to the code length [10], and its consistency with the accompanied program.

Typed Assembly Language, as a variant of Proof-Annotating Code, restricts its safety guarantees to type and memory safety as well as the legal control flow of assembly programs, and stresses the translation of type-correct programs of the source language to type-correct assembly code [52]. The restricted scope of the verification process allows to automatically generate the safety certificate, in the form of type annotations, on the producer side of the corresponding mobile code system. Furthermore, the generation of a safety proof, as required by Proof-Carrying Code, is omitted, since verification on the consumer side is done by a type-checker which utilizes the annotated type information of the transmitted assembly code.

It should be noted, that the three presented safety concepts (i.e., verification based on data-flow analysis, implicitly legal program encodings, and certifying compilers) are orthogonal and may be combined in several ways. All of these concepts rely on representing only programs translated from a safe source language, and—with the exception of the Common Intermediate Language's unmanaged extensions for unverified programs—unsafe features like pointer arithmetic are not supported by any of the presented intermediate representations of mobile code.

5 Conclusion

In the paper, we have provided an overview of common intermediate representations of mobile code, discuss the strengths and weaknesses of each, and compare its properties with that of the other representations.

The comparison of different intermediate representations (see Table 1) leads us to the conclusion that there is no unqualified 'best' mobile code format. One reason for this may be that due to a tendency to focus on one single aspect of the mobile code framework. For example, the developers of PCC were mostly concerned with providing increased security but did not address portability. On the other hand, developers of ANDF provided a very portable distribution format but did not address advanced safety requirements.

Instead, it is obvious that for each intermediate representation, there are disadvantages, which cause it to fail to live up to the ideal. As a consequence, except for Microsoft's CIL representation, none of the suggested mobile code formats can be seen as a serious commercial challenger for Java's Bytecode format. This is also supported by the observation that, except for Java Bytecode and CIL, for none of the other intermediate representations a mobile code system other then of prototype status has been developed.

Because of the wide acceptance of Java Bytecode and because none of the alternative intermediate representations is the ne plus ultra, most current mobile code projects have shied away from the developing of novel intermediate representations. Instead, recent research projects in that area attempt to improve the JVM [20] or integrate features of some of the representations into Java Bytecode. In particular, representatives of this trend are projects that adapt the concepts of Proof-Carrying Code and type-separation to Java Bytecode [2, 26, 72].

Acknowledgment

This investigation has been supported in part by the Deutsche Forschungsgemeinschaft (DFG) under grants AM-150/1-1 and AM-150/1-3.

References

[1] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'1996)*, volume 31 of *ACM SIGPLAN Notices*, pages 127–136, New York, May 1996. ACM Press.

[2] P. Adler and W. Amme. Improving the java virtual machine using type-separated bytecode. In *Proceedings of the Workshop on Compilers for Parallel Computers (CPC'2006)*, pages 256–263, Jan. 2006.

[3] W. Amme, N. Dalton, M. Franz, and J. von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*, volume 36 of *ACM SIGPLAN Notices*, pages 137–147, Snowbird, Utah, USA, June 2001. ACM Press.

[4] W. Amme, J. von Ronne, and M. Franz. Ssa-based mobile code: Implementation and empirical evaluation. Technical Report CS-TR-2006-005, Computer Science, The University of Texas at San Antonio, 2006.

[5] A. W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 247–256, Boston, MA, USA, June 2001. IEEE Computer Society Press.

[6] E. Armstrong. Cover story: HotSpot: A new breed of virtual machine. *JavaWorld: IDG's magazine for the Java community*, 3(3), Mar. 1998.

[7] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA'2000)*, volume 35 of *ACM SIGPLAN Notices*, pages 47–65, New York, Oct. 2000. ACM Press.

[8] F. Broustaut, C. Fabre, F. de Ferrière, É. Ivanov, and M. Fiorentini. Verification of ANDF components. *ACM SIGPLAN Notices*, 30(3):103–110, Mar. 1995.

[9] J. Bundgaard. An andf based ada 95 compiler system. In *TRI-Ada '95: Proceedings of the conference on TRI-Ada '95*, pages 436–445, New York, NY, USA, 1995. ACM Press.

[10] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2000)*, volume 35 of *ACM SIGPLAN Notices*, pages 95–107, New York, June 2000. ACM Press.

[11] C. Colby, G. C. Necula, and P. Lee. A proof-carrying code architecture for Java. In *Proceedings of the International Conference on Computer Aided Verification (CAV'2000)*, June 2000.

[12] R. Crelier. OP2: A Portable Oberon Compiler. Technical Report 1990TR-125, Swiss Federal Institute of Technology, Zürich, Feb. 1990.

[13] I. F. Currie. *TDF Specification, Issue 4.0.* Defence Research Agency, England, June 1995.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[15] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 41–49, New York, NY, USA, 2003. ACM Press.

[16] M. de Icaza and B. Jepson. Mono and the .Net framework. *Dr. Dobb's Journal of Software Tools*, 27(1):21–24, 26, Jan. 2002.

[17] J. des Rivières and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[18] D. R. Engler. VCODE : A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'1996)*, volume 31 of *ACM SIGPLAN Notices*, pages 160–170, New York, May 1996. ACM Press.

[19] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, Apr. 1967. American Mathematical Society.

[20] B. Folliot, I. Piumarta, L. Seinturier, C. Baillarguet, C. Khoury, A. Leger, and F. Ogel. Beyond flexibility and reflection: The virtual virtual machine approach. In D. Grigoras, A. Nicolau, B. Toursel, and B. Folliot, editors, *IWCC*, volume 2326 of *Lecture Notes in Computer Science*, pages 16–25. Springer, 2001.

[21] M. Franz. Emulating an operating system on top of another. *Software – Practice and Experience*, 23(6):677–692, June 1993.

[22] M. Franz. *Code-Generation On-the-Fly: A Key for Portable Software*. PhD thesis, Institute for Computer Systems, ETH Zürich, 1994.

[23] M. Franz. The Java Virtual Machine: A passing fad? *IEEE Software*, 15(6):26–29, Nov. / Dec. 1998.

[24] M. Franz. Open standards beyond java: On the future of mobile code for the internet. *J. UCS*, 4(5):522–533, 1998.

[25] M. Franz, W. Amme, M. Beers, N. Dalton, P. H. Frohlich, V. Haldar, A. Hartmann, P. S. Housel, F. Reig, J. von Ronne, C. H. Stork, and S. Zhenochin. *Making mobile code both safe and efficient. In Foundations of Intrusion Tolerant Systems*, pages 337–356. IEEE Computer Society Press, 2003.

[26] M. Franz, D. Chandra, A. Gal, V. Haldar, C. W. Probst, F. Reig, and N. Wang. A portable virtual machine target for proof-carrying code. *Journal of Science of Computer Programming*, 57(3):275–294, Sept. 2005.

[27] M. Franz and T. Kistler. Introducing juice. Published in Internet, 1996.

[28] M. Franz and T. Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, Dec. 1997.

[29] M. Franz, C. Krintz, V. Haldar, and C. H. Stork. Tamper proof annotations. Technical Report 02-10, Department of Information and Computer Science, University of California, Irvine, Mar. 2002.

[30] M. Franz and S. Ludwig. Portability redefined. In *Proceedings of the 2nd International Modula-2 Conference*, Loughborough, England, Sept. 1991.

[31] A. Gampe. An interpreter for safetsa. Master's thesis, 2006. Masters thesis, Friedrich-Schiller-University, Jena, Germany.

[32] F. Gasperoni and G. Dismukes. Multilanguage programming on the JVM: The Ada 95 benefits. *ACM SIGADA Ada Letters*, 20(4):3–28, Dec. 2000. Special Issue: Presentations from SIGAda 2000.

[33] J. Gosling. Java intermediate bytecodes. In *Proceedings of the Workshop on Intermediate Representations (IR'1995)*, volume 30 of *ACM SIGPLAN Notices*, pages 111–118, San Francisco, CA, Jan. 1995. ACM Press.

[34] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Reading, MA, USA, second edition, 2000.

[35] K. J. Gough. Stacking them up: a comparison of virtual machines. In *Proceedings of the 6th Australasian conference on Computer systems architecture*, pages 55–61. IEEE Computer Society Press, Feb. 2001.

[36] D. Grossman and G. Morrisett. Scalable certification of native code: Experience from compiling to TALx86. Technical Report TR2000-1783, Cornell University, Computer Science, Feb. 2000.

[37] D. Grossman and J. G. Morrisett. Scalable certification for typed assembly language. In R. Harper, editor, *TIC*, volume 2071 of *Lecture Notes in Computer Science*, pages 117–146. Springer, 2000.

[38] D. Grossman, J. G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Trans. Program. Lang. Syst*, 22(6):1037–1080, 2000.

[39] V. Haldar, C. H. Stork, and M. Franz. The source is the proof. In *The 2002 New Security Paradigms Workshop*, pages 69–74, Virginia Beach, VA, USA, Sept. 2002. ACM SIGSAC, ACM Press.

[40] B. S. Hansen and J. U. Toft. The formal specification of ANDF, an application of action semantics. In *Proceedings of the 1st International Workshop on Action Semantics, Edinburgh, 1994*, number NS-94-1 in BRICS Notes Series, pages 34–42. BRICS, Dept. of Computer Science, Univ. of Aarhus, 1994. BRICSreportNS941.

[41] K. Hansson. Java: Trees versus bytes. Master's thesis, a BComp Honours thesis, 2004.

[42] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, Jan. 1993.

[43] A. Hartmann, W. Amme, J. von Ronne, and M. Franz. Code annotation for safe and efficient dynamic object resolution. In J. Knoop and W. Zimmermann, editors, *Proceedings of Compiler Optimization Meets Compiler Verification (COCV'2003)*, pages 18–32, Warsaw, Poland, Apr. 2003.

[44] IBM Research. *Jikes RVM User's Manual*, v2.0.3 edition, Mar. 2002.

[45] R. Keskar and R. Venugopal. *Compiling safe mobile code. In Compiler Design Handbook: Optimzations and machine code generation*, pages 763–800. CRC Press, 2003.

[46] T. Kistler and M. Franz. Slim binaries. techreport 96-24, Department of Information and Computer Science, University of California, Irvine, June 1996.

[47] T. Kistler and M. Franz. A Tree-Based alternative to Java byte-codes. *International Journal of Parallel Programming*, 27(1):21–34, Feb. 1999.

[48] T. P. Kistler. Continuous program optimization. *PhD Dissertation, University of California, Irvine*, 1999.

[49] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, second edition, 1999.

[50] E. Meijer, R. Wa, and J. Gough. Technical overview of the common language runtime. Microsoft, Oct. 2000.

[51] 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 *2nd ACM SIGPLAN Workshop on Compiler Support for System Software (WCSSS'99)*, pages 25–35, Atlanta, GA, USA, May 1999.

[52] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[53] R. Nagy. Menu in activeX controls, Jan. 08 2004.

[54] G. C. Necula. Proof-carrying code. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'1997)*, ACM SIGPLAN Notices, pages 106–119, New York, NY, USA, Jan. 1997. ACM Press.

[55] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, Sept. 1998. Technical report CMU-CS-98-154.

[56] G. C. Necula and P. Lee. Research on proof-carrying code for untrusted-code security. In *Proceedings of the Conference on Security and Privacy (S&P'1997)*, pages 204–204, Los Alamitos, May 1997. IEEE Computer Society Press.

[57] G. C. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Proceedings of the Annual Symposium on Logic in Computer Science (LICS'1998)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.

[58] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 39(4):612–625, 2004.

[59] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 142–154, New York, NY, USA, 2001. ACM Press.

[60] K. V. Nori, U. Ammann, K. Jensen, N. Nageli, and C. Jacobi. Pascal-P implementation notes. In D. W. Barron, editor, *Pascal – The Language and its Implementation*, pages 125–170. John Wiley & Sons, Ltd., 1981.

[61] OpenGroup. Architecture Neutral Distribution Format (XANDF) Specification. *Open Group Specification P527*, page 206, Jan. 1996.

[62] M. Paleczny, C. A. Vick, and C. Click. The java hotspot^{TM} server compiler. In *Java^{TM} Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[63] E. Schanzer. Performance considerations for run-time technologies in the .net framework. Microsoft technical report, Microsoft Corporation, Aug. 2001.

[64] Z. Shao. An overview of the FLINT/ML compiler. In *Proceeding of the Workshop on Types in Compilation (TIC'1997)*, ACM SIGPLAN Notices, Amsterdam, The Netherlands, June 1997. ACM Press.

[65] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl. Virtual machine showdown: stack versus registers. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 153–163, New York, NY, USA, 2005. ACM Press.

[66] K. Sohr. *Die Sicherheitsaspekte von mobilem Code*. PhD thesis, Universität Marburg, 2001.

[67] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification and Validation*. Springer, 2001.

[68] A. Stump and D. L. Dill. Faster proof checking in the Edinburgh Logical Framework. In *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 392–407. Springer-Verlag, July 2002.

[69] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, Sept. 1997.

[70] J. von Ronne, W. Amme, and M. Franz. Safetsa: An inherently type-safe ssa-based code format. Technical Report CS-TR-2006-004, Department of Computer Science, The University of Texas at San Antonio, 2006.

[71] T. A. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, June 1984.

[72] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'2005)*, pages 16–30, 2005.