Efficient Code Certification

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

January 8, 1998

Abstract

We introduce a simple and efficient approach to the certification of compiled code. We ensure a basic but nontrivial level of code safety, including control flow safety, memory safety, and stack safety. The system is designed to be simple, efficient, and (most importantly) relatively painless to incorporate into existing compilers. Although less expressive than the *proof carrying code* of Necula and Lee or *typed assembly language* of Morrisett et al., our certificates are compact and relatively easy to produce and to verify. Unlike JAVA bytecode, our system operates at the level of native code; it is not interpreted and no further compilation is necessary.

1 Introduction

An exciting recent development in program verification is the notion of *certified code*. When downloading executable code from an untrusted foreign source, a client also downloads a certificate that is checked before running the code to ensure that it is safe to run locally. The certificate is produced by the supplier at compile time and verified by the client at load time; both operations are (or should be) automatic and invisible to both parties.

This notion first appeared in a practical system in the JAVA programming language. The JAVA compiler produces virtual machine instructions that can be verified by the client before execution and that are meant to provide a certain basic level of security [2]. Despite certain failings in the initial implementation [1], it nevertheless constituted a significant step forward in practical programming language security. It not only introduced a simple and effective approach to providing a basic level of security, but more importantly, it helped to galvanize the attention of the programming language and verification community on critical security issues engendered by the rise of the Internet.

Recent work by Necula and Lee [5, 6] on proof carrying code (PCC), by Morrisett et al. [4] on typed assembly language (TAL), and by Schneider [8, 9] on automata-based security have demonstrated that general principles of logic, type theory, and program verification can be brought successfully to bear on the security problem. These systems are quite general and can potentially be used to express and verify a variety of safety properties. Schneider [8, 9] identifies the class of safety properties that can be verified by these methods.

In the case of TAL, typing information from a high-level program written in a strongly-typed language is carried through a series of transformations through a platform-independent typed intermediate language (TIL) [3, 10] and finally down to the level of the object code itself. The resulting type annotation can be checked by an ordinary type checker. In the case of PCC, correctness properties are expressed in first-order logic, and a proof is constructed based on a general system such as Edinburgh LF and provided along with the program. Although these approaches are quite expressive and general, their chief drawback is that the certificates they produce tend to be large and time-consuming to generate and to verify. For example, proofs in PCC are from 3 to 7 times the size of the object code [5, 6]. The automata-based approach of Schneider [8, 9] requires a runtime call on an automaton for each executed instruction, unless an optimizer can determine that the call is unnecessary; thus without effective optimization, runtime performance could be seriously degraded as well.

In contrast with these approaches, our goal with the present work is to limit ourselves to a fixed and relatively basic level of security, but to provide it as simply, efficiently, compactly, and invisibly as possible. We are able to ensure

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

These rough descriptions are not to be taken as formal definitions. A more rigorous treatment can be found in Section 3. In that section, we define formally the meaning of *control flow safety*, *memory safety*, and *stack safety*. These safety conditions are mutually dependent in the sense that *none of them are safe unless all of them are safe*. Thus, the level of safety we provide is, in an informal sense, the minimal nontrivial level of safety ensuring any one of these conditions.

Our foremost concerns are performance and ease of implementation. We are willing to sacrifice the generality of TAL or PCC, the platform-independence of JAVA bytecode, and the enhanced expressiveness of Büchi automata for a more streamlined system tailored to the platform at hand. Our motivation is that, although the demand for security is great, the technology will not be adopted routinely in industrial-grade compilers unless it is relatively painless to do so and entails no significant performance degradation.

Although our approach is necessarily highly platform-dependent, it is still possible to isolate the logical principles at work. We do this in Section 3 by formulating and proving a soundness theorem. The theorem can be applied to show that the conditions checked by our verifier are sufficient to ensure control flow safety, memory safety, and stack safety.

Although inspired by PCC, it would be inaccurate to call our certificates *proofs*, because they are not proofs in any formal system. They are more accurately described as *structured annotations* that merely direct the verification process. Guided by this structural information, the verifier checks a simple set of conditions that are sufficient to imply the desired safety properties. As such, the system is closer in spirit to the JAVA bytecode verifier, although it operates at the level of native code. With PCC, the safety conditions are expressed explicitly and proved by the compiler as part of the implementation; this becomes difficult in the presence of loops, since loop invariants cannot be generated automatically in general. In contrast, our system generates relatively simple annotations from information already present as part of the compilation process. The proof that the code is safe

```
(define-method append (x y)
    (if (null? x)
        y
        (pair (head x) (append (tail x) y))))

(append (list 1 2 3) (list 4 5))

==> (1 2 3 4 5)
```

Figure 1: Appending two lists

to run need not appear as part of the implementation, but can be done on paper once and for all.

The centerpiece of this initiative is a working prototype that we have built to demonstrate the feasibility of this approach. The prototype is virtually complete (six months ahead of schedule!) and we describe it in Section 4. The construction of the prototype constitutes the first year of a three-year project. In the remaining two years, we hope to see to what extent this technology can be incorporated in industrial-grade compilers.

2 How It Works

In this section we describe the form of the certificate and how it obviates the need for a full first-order proof as with PCC or a complete type annotation as with TAL.

2.1 Block Structure

The code generated by a compiler has a natural context-free structure that mirrors the structure of the high level program from which it was compiled. This is inescapable, since programming languages are typically defined by context-free grammars and compilation is a recursive process that acts on the parse tree. The key idea is to exploit this natural structure to direct the verification.

This structure consists of well-nested intervals of instructions, called *blocks*, each of which is generated by the compiler for a specific purpose. For example, a *program block* contains the compiled code for the main program; a *call block* contains the compiled code for the body of a function; and an *eval block* contains code for evaluating an expression. The compiler can identify this block structure and produce the block tree as part of the certificate.

For example, consider the program of Figure 1 for appending two lists. Figure 3 on p. 17 shows the block structure of the compiled code. For example, the block

```
;;;evaluate (pair (head x) (append (tail x) y))
.begin eval block
     ;;;evaluate 2 argument(s)
     ;;;evaluate (head x)
     .begin eval block
          .
          . [ code to evaluate (head x) ]
          .
     .end eval block type= ? status= register location= eax
     C26: push eax                ;push onto stack
     ;;;evaluate (append (tail x) y)
     .begin call sequence
          .
          . [ code to evaluate (append (tail x) y) ]
          .
     .end call sequence args= 2 type= PAIR status= register location= eax
     ;;;apply pair
     C55: pop ebx                 ;head of pair
     C56: alloc ecx,3             ;new cons cell
     C57: mov [ecx],PAIR          ;save runtime type of pair
     C58: mov [1+ecx],ebx         ;save head
     C59: mov [2+ecx],eax         ;save tail
.end eval block type= PAIR status= register location= ecx
```

Figure 2: Residue of an eval block

[1,72) labeled "recursive method initialization" contains the definition of append. The subblock [20,60) labeled "eval block" contains code to evaluate the else clause of the body of the function, namely `(pair (head x) (append (tail x) y))`.

2.2 Residues

The *residue* of a block is the set of instructions remaining after all subblocks have been deleted. The code in the residue of a block is called *residual code*. For example, Figure 2 shows the residue of the block [20,60) mentioned above. Residues are typically small and there only a small finite number of them. The residue in Figure 2 contains only six instructions, which perform the pair operation once the arguments have been evaluated.

The lines starting with `.begin` and `.end` are annotations and are part of the certificate. They define the boundaries of the blocks and give basic typing information that is needed for verification.

2.3 Control Flow

There are three control flow mechanisms:

(i) conditional and unconditional jumps;

(ii) **call** and **return**; and

(iii) fallthrough.

The instructions **call** and **return** have the side effect of manipulating the stack; they push and pop the return address. A *fallthrough* is the flow of control to the instruction immediately following the one just executed. An instruction that can fall through to the next instruction is called a *fallthrough instruction*. Most instructions are fallthrough instructions; examples of those that are not are **halt**, **call**, **return**, and unconditional jumps. Conditional jumps are always considered fallthrough instructions.

Our verifier checks that control flow respects block structure: no control flow instruction (conditional and unconditional jump, **call**, or **return**) or fallthrough may traverse a block boundary except in a few strictly prescribed ways. These restrictions ensure that residual code is executed with a certain degree of atomicity; for example, it will be impossible for there to be an unexpected jump to the middle of a sequence of residual instructions from outside the residue. This is crucial for ensuring memory safety: without it, we could not be certain that registers contain correct pointers and data values when writing to memory.

A program block may only be entered via a jump to the first location and may not exit except via a **halt** instruction. A call block may not be entered or exited via a jump or fallthrough, but only via **call** to the first location and **return** or **halt**. All other types of blocks must be entered via a jump or fallthrough to the first instruction of the block and must be exited via a jump or fallthrough to the instruction immediately following the block (not counting subroutine calls from within the block). The residue of a non-call block may not contain a **return**. Conditions reminiscent of these appear in the JAVA bytecode verifier [2].

These are straightforward and natural restrictions. They are satisfied by our compiler, and we suspect that they (or something very similar) are satisfied by compilers in general, or can be made to be satisfied fairly easily.

These restrictions alone are not sufficient to ensure control flow safety; we must have stack safety and memory safety as well. For instance, if the subblock [27,55) in Figure 2 does not preserve the stack, then there is no guarantee that the object popped at C55, which is supposed to be the same as the object pushed at C26, is even a valid data object. If that object happened to be a function, this could result in an invalid call later on.

For example, consider a subroutine call with two arguments. Although the call sequence could be arbitrarily long depending on the complexity of the arguments that need to be evaluated, the residue only contains the two pushes, the call, and the two pops. A random jump into this residue from outside would almost certainly corrupt the stack. However, the control flow restrictions ensure that this cannot happen. The block can be annotated with the information that this is a subroutine call with two arguments, and the verifier need only check that the residue contains two pushes and two pops in the correct order.

Further control flow conditions are checked in Section 2.6. In particular, all **call** instructions must occur in the context of the standard subroutine linkage or exception handling mechanism. This will guarantee, among other things, that every **call** targets the first instruction of a call block.

2.4 Stack Safety

For stack safety, the verifier needs to check that in any residue, pushes and pops occur in well-nested pairs, and that pushes are always executed before their corresponding pops. This might ordinarily be done using flow analysis on the control flow graph [7]. However, with the block structure explicit, no expensive analysis is necessary. Pushes and pops are generated only in well-defined contexts, usually as part of a standard subroutine call linkage. A simple check of the residual code of the linkage can ensure that all pushes and pops match.

2.5 Memory Access

Memory must be accessed correctly. Every memory operation, read or write, has a precondition for safe execution, and we must guarantee that the precondition is met whenever that instruction is executed.

Our implementation uses a dedicated environment register e, and the environment only changes at a subroutine call. For each call block, we must check that

(i) the old environment pointer is saved at the beginning and restored at the end of the call block;

(ii) storage for the new environment table is allocated and initialized correctly, and e is loaded with the address of the new table; and

(iii) all accesses off register e (except in sub-call blocks) are at a constant offset not exceeding the length of the environment table.

The length of the environment table is known at compile time and is included in the call block annotation. This information can be used to check (iii). Conditions (i) and (ii) can be checked simply by comparing the residual code to standard call linkage code obtained by table lookup.

We must also check that all storage allocated off the system heap is initialized before it is used. We assume that the language provides a default value for variables of all types (as does JAVA) and that the compiler initializes newly allocated storage immediately. This can be verified by table lookup of the residual code. For example, in the residue of Figure 2, we must check that the new cons cell allocated in line C56 is immediately initialized, which it is; this is done by table lookup to compare the residual code to the standard code for an inline application of pair.

2.6 Residual Code

Endpoints of blocks are often annotated with extra typing information. For example, the annotation at the end of the block illustrated in Figure 2 says that register `ecx` contains a valid pair object. The residual code of the block must correctly guarantee that these conditions are met whenever control is at that point, under the inductive assumption that subblocks and calls satisfy memory and stack safety and guarantee their annotations as well. Then at the very worst, the verifier need only check by table lookup that the correct residual code is there. Often even simpler checks suffice.

Some assumptions need not be explicitly represented in the annotation. For example, in our implementation, methods are always called with a pointer to the method object in register `eax`, which the method initialization code depends on. Because of control flow safety, the only way a method can be called is from within a *call sequence*, which is just another type of block. Since any valid call sequence guarantees the desired property, the verifier need only check the residual code of all call sequences.

3 Soundness

In this section we prove a soundness theorem. The theorem characterizes a collection of basic safety properties and establishes sufficient conditions that imply them and that can be checked by a verifier. The most difficult part of this theorem is not its proof but its formulation. Once the theorem is stated, the proof is a relatively straightforward inductive argument.

The chief difficulty in the formulation lies in the conflicting desires of establishing the correctness of our implementation and more general applicability. We have found that a sufficiently rigorous proof of correctness of our implementation depends on various implementation-specific assumptions. Although we would like the argument presented here to establish correctness of our implementation, we would also hope that it could be applied mutatis mutandis to a variety of situations. We have therefore formulated the theorem in a fairly general way under a few reasonable assumptions. We are encouraged by the fact that there is considerable flexibility in both compiler implementation and in the form of the block structure and verification conditions. If we impose restrictions on the structure of object code that are not currently satisfied by some compiler, there is room for movement on both sides. The reengineering of an existing code generator or optimizer to make them respect block structure is not an insurmountable task and may even be worthwhile for independent reasons.

We remark in passing that it would be difficult to formulate this theorem solely in terms of types. Although types do enter heavily into the formulation, it is difficult to characterize some of the basic safety properties we are interested in. For example, "the stack is preserved across a subroutine call" is difficult to express purely type-theoretically. The difficulty lies in the lack of a natural means to compare the values of variables in two distinct states related by a runtime execution path. Types and set constraints are very good for talking about the static properties of an object or relations between objects at a particular point in the execution, but not about the relationship between objects at different points in the execution.

3.1 Assumptions

Before we state our soundness theorem, we start with some basic assumptions.

We assume that there are two dedicated registers, namely the *environment pointer* e and the *stack pointer* s. The stack is used to hold arguments to functions, to preserve e across calls, and to save return addresses. We assume that the stack pointer s always points to the top element of the stack and that it is initialized by the system before calling the user code. We assume for simplicity that s is affected only by the instructions **push**, **pop**, **call**, and **return**; no explicit register operations involving s are allowed. We also assume for simplicity that there is no danger of stack overflow. In a real system, there would need to be some mechanism, either hardware or software, for checking for stack overflow, and the verifier would need to ensure that that mechanism was in place.

In addition, we assume that there is a mechanism for garbage collection that periodically and invisibly reclaims heap storage no longer accessible. In our implementation, allocation of heap memory is effected by an atomic **alloc** instruction. Otherwise, we assume that there are no external agents that can change the state of memory.

We assume that the stack, heap, program, and constant data all occupy disjoint memory segments. Stack memory is contiguous, and the stack grows in one direction (in our implementation, from high memory downward).

3.2 Scoping Discipline

Our high-level language is statically scoped. This means that each method (function) is executed in an *environment* determined by the bindings that are in effect when the method is initialized. This environment is used to resolve the references to free identifiers in the body of the method. In our implementation, when a method object is initialized, a local copy of each free identifier occurring in the body is created and initialized with the current value obtained from the enclosing environment. This local environment is associated with the method throughout its lifetime.

This implementation may seem inefficient due to the cost of creating the local environment when the method is initialized. But by amortizing the initialization cost over the lifetime of the method object, it in fact works out to constant time per access, provided for each free identifier there is at least one call that accesses that identifier.

The environment pointer e is initialized during program initialization and always points to the current environment. Thereafter, e is saved on the stack and restored by the standard subroutine linkage before and after any call.

Other scoping disciplines are certainly possible and may even be more complicated, as for example with object oriented languages. However, the basic technique for dealing with changing environments is already exercised at this level, and we believe that it can serve as a paradigm for more complicated situations.

3.3 A Soundness Theorem

Suppose we have determined a block structure with annotations that specify the type of each block as well as any associated pre- or postconditions. Suppose further that the blocks are well-nested, the outermost block is a program block containing all code, and the restrictions on control flow as specified in 2.3 are satisfied; that is,

(CF1) a program block may only be entered via a jump to the first instruction and may only be exited via a **halt** instruction (not counting subroutine calls from within the block);

(CF2) a call block may not be entered via a jump or fallthrough and may only be exited via a **halt** or **return** (not counting subroutine calls from within the block); and

(CF3) for any other type of block, all jumps or fallthroughs entering the block must target the first instruction of the block, and the block must be exited via a jump or fallthrough to the instruction immediately following the block (not counting subroutine calls from within the block).

Definition 3.1 A memory operation (read or write) is *safe* if it references either (i) a location in heap storage that has previously been allocated, (ii) a valid location on the runtime stack, or (iii) a valid location in the program's constant data area. □

Definition 3.2 An *execution* of a block is a computation sequence beginning at the first instruction of the block in some state satisfying the precondition of the block. □

Definition 3.3 An execution of a block is *safe* if it satisfies the following properties.

(S1) If and when control exits the block (excluding subroutine calls), it does so in an acceptable way as specified by (CF1)–(CF3).

(S2) Any **call** is to the first instruction of a call block.

(S3) Registers s and e and the stack are preserved; that is, they are in the same state on block exit as they were on entry.

(S4) The postcondition of the block is satisfied on exit.

(S5) All memory references are safe.

□

Definition 3.4 The *standard assumption* for a block is that

(SA1) the precondition of the block is satisfied on entry;

(SA2) execution begins at the first instruction; and

(SA3) the execution of any subblock or called subroutine is safe.

□

Definition 3.5 We say that a block is *residually safe* if

(LS1) under the standard assumption, the residual code guarantees that the precondition of any subblock is satisfied upon entry to that subblock;

(LS2) under the standard assumption, the residual code guarantees that when control is at the site of any **call** instruction in the residue, the target of the **call** is the first instruction of a call block, and the precondition of the call block is satisfied (this may include constraints on the number, type, and location of arguments);

(LS3) ignoring subblocks and subroutine calls, the residue preserves s, e, and the stack;

(LS4) under the standard assumption, the residual code guarantees that the postcondition of the block is satisfied on exit;

(LS5) under the standard assumption, the residual code guarantees that all memory references in the residue are safe.

□

The following is our main theorem.

Theorem 3.6 *If all blocks are residually safe, then all executions of all blocks are safe. In particular, if the outermost program block is executed starting at the first location in a state satisfying its precondition, then the entire program is safe.*

The safety of an execution is a dynamic notion, but the residual safety of a block is a static notion that can be checked by a verifier. We will prove the theorem formally below, but first let us illustrate the idea with an example. Consider the block of Figure 2, an eval block that creates a PAIR object. The residual code of the block is given explicitly; not shown are the two subblocks that evaluate the head and the tail of the new PAIR object.

Instruction C59 stores a pointer to the tail of the pair in a newly allocated cons cell. This is a memory access, and we must verify that it is safe (LS5). By (CF1)–(CF3), which are verified separately, there are no jumps with target between C56 and C59 originating from outside the block or from within any subblocks, and by inspecting the residual code it can be seen that there are no such jumps from the residue either; therefore the only computation path leading to C59 is through C56, C57, and C58. This guarantees that the memory reference in C59 is to the cons cell just allocated.

For (LS4), the postcondition of the block states that register `ecx` contains a pointer to a valid PAIR object. By definition, this means that `ecx` points to a heap cell of length 3 containing the runtime type of a PAIR and pointers to two valid data objects, the head and the tail. The preconditions of the subblocks that evaluate the head and the tail are satisfied vacuously (this is (LS1)), therefore by clause (SA3) of the standard assumption, the executions of these subblocks are safe. By the safety condition (S4) for these subblocks, the objects they produce are valid data objects, and they are returned in register `eax`. The head, which is evaluated first, is pushed onto the stack while the tail is being evaluated; since the subblock evaluating the tail satisfies (S3), the head is still there on top of the stack after the tail is evaluated.

Property (LS3) holds because the residue does not mention e or s, and there is one **push**/**pop** pair in the correct order. Properties (LS1) and (LS2) are satisfied vacuously.

In this case, the easiest way for the verifier to verify all these conditions is to compare the residual code of the block to the standard code for creating a PAIR object produced by the compiler, which the verifier knows about. The annotation of the block specifies a name that identifies the code that should be there, and the verifier checks by table lookup that the correct code is indeed there.

Proof of Theorem 3.6. The proof of the theorem is by induction on the length of executions and the number of times subblocks are executed and call blocks are called.

Assume that all blocks are residually safe, and consider the execution of any block B. By definition, the execution begins at the first location of B in some state satisfying B's precondition; thus clauses (SA1) and (SA2) of the standard assumption hold. We also have (SA3) for executions of subblocks and called call blocks by the induction hypothesis. Thus the standard assumption holds for B.

Since B is residually safe by assumption, as long as the execution remains in the residue of B, by (LS5) all memory references executed in the residue are safe. By assumptions (CF1)–(CF3), the execution must remain in the residue of B until either

(i) it falls through or jumps to the first instruction of a subblock;

(ii) it executes a **call** instruction; or

(iii) it exits B.

In case (i), since B is residually safe, it satisfies (LS1), therefore the precondition of the subblock holds. By the induction hypothesis, the execution of that subbblock is safe, therefore satisfies (S1)–(S5). In particular, if and when the subblock exits, it does so in a state satisfying its postcondition, and execution of the residue of B resumes at the instruction immediately following the subblock.

In case (ii), since B is residually safe, it satisfies (LS2), therefore the precondition of the call block holds. By the induction hypothesis, the execution of that call block is safe, therefore satisfies (S1)–(S5). In particular, by (S1) and (S3), the call block returns to the instruction immediately following the **call** in a state satisfying the postcondition of the call block, and execution of the residue of B resumes there.

In case (iii), since B is residually safe, it satisfies (LS4) upon exit.

Cases (i) and (ii) may occur arbitrarily many times before case (iii). Indeed, case (iii) may never occur.

Now we argue that B satisfies (S1)–(S5). It satisfies (S1) by assumptions (C1)–(C3). It satisfies (S2) because of (LS2). It satisfies (S3) because of (LS3) and the inductive assumption that all executions of subblocks and called call blocks satisfy (S3). It satisfies (S4) because of (LS4). Finally, it satisfies (S5) because of (LS5) and the inductive assumption that all executions of subblocks and called call blocks satisfy (S5). $\square$

4 Project Status

We have built a working prototype to illustrate the feasibility of this approach. The prototype comprises the first year of a three-year project as part of a broader initiative on language-based security. The system consists of

- a compiler for a statically scoped SCHEME-like functional language that produces something resembling flat 32-bit x86 (Pentium) assembly code and a certificate;
- a loader that loads the compiled object and preprocesses the certificate;
- an x86 emulator that executes the object interpretively; and
- a verifier.

The system is publicly available from the Cornell Computer Science Department's web site `http://www.cornell.edu/`.

5 Other Issues

Optimization For the future, we would like to investigate ways to handle compiler optimizations in the presence of annotations. Each time an optimization is performed, a transformation of the certificate would also have to be performed, and it is an interesting question how to do this in a way that preserves the semantics of the code and the certificate. Some simple local optimizations such as inlining and tail recursion elimination can already be handled easily as special cases. We would like to investigate methods for handling other optimizations as well.

Strong typing vs. runtime checks Our approach works only for type-safe languages. Our prototype compiler is for a statically scoped SCHEME-like functional language in which some types are inferred, but in general types are dynamic and runtime checks are used where necessary to insure type safety. In a strongly typed language or with an optimizing compiler, many of these checks would be unnecessary. In such languages, the extra type information or information obtained from program flow analysis can be included in the certificate.

Necula and Lee [5, 6] argue that PCC can be used to eliminate certain runtime type checks. This may be true, but to our mind it is an orthogonal issue. For the purposes of verification of safety properties, it is irrelevant whether the required type information comes down from a typing of the source code, from program analysis, or from runtime type checks. The compiler knows for one reason or another that the program is type safe, and it is our task to figure out how to make that reason explicit in the certificate.

Dynamic loading One issue that appears not to present a serious problem is the dynamic loading and linking of libraries. The library can also contain a certificate that can be verified in the same way when it is first loaded if necessary.

Storage management This is also not a problem. The garbage collector is assumed to be part of the local runtime system (*not* part of the user program—another reason why this approach would not work for C) and therefore trusted. Memory safety will prevent the program from accessing garbage-collected storage.

Other worries Many thorny complications exist in industrial-grade compilers, such as nonlocal jumps, exception handling, aliasing, and concurrency. It remains to be seen how effectively the technology described in this paper will scale.

Acknowledgements

I am indebted to Greg Morrisett for introducing me to the problem and for his generous help and constant encouragement. This work has benefited in countless ways from his deep insight and encyclopedic knowledge of programming languages. Fred Schneider read an earlier draft of this paper and suggested many improvements to the presentation. The participants of the MILC seminar at Cornell have provided an exciting and supportive atmosphere and have contributed greatly to the development of these ideas. Jason Hartline provided a fine preliminary implementation of the emulator.

References

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

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

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

[4] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *1998 Symposium on Principles of Programming Languages*. IEEE, January 1998. To appear.

[5] George C. Necula. Proof-carrying code. In *Proc. 24th Symp. Principles of Programming Languages*. ACM, January 1997.

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

[7] Thomas W. Reps. Program analysis via graph reachability. In J. Maluszynski, editor, *Proc. ILPS '97: International Logic Programming Symposium*, pages 5–19. MIT Press, October 1997.

[8] Fred B. Schneider. Enforceable security policies, September 1997. Preprint.

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

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

```
0 begin program block
| 0 begin call block
| | 1 begin recursive method initialization
| | | 4 begin call block
| | | | 8 begin method body
| | | | | 8 begin eval block
| | | | | | 8 begin eval block
| | | | | | 18 end eval block type= 306 status= 401 location= eax
| | | | | | 20 begin eval block
| | | | | | | 20 begin eval block
| | | | | | | 26 end eval block type= 308 status= 401 location= eax
| | | | | | | 27 begin call sequence
| | | | | | | | 42 begin eval block
| | | | | | | | 48 end eval block type= 308 status= 401 location= eax
| | | | | | | 55 end call sequence args= 2
| | | | | | 60 end eval block type= 304 status= 401 location= ecx
| | | | | 62 end eval block type= 308 status= 401 location= ecx
| | | | 62 end method body
| | | 66 end call block free= 1 bound= 2
| | 72 end recursive method initialization fns= 1
| | 72 begin call sequence
| | | 87 begin eval block
| | | | 89 begin eval block
| | | | | 91 begin eval block
| | | | | 99 end eval block type= 304 status= 401 location= ecx
| | | | 104 end eval block type= 304 status= 401 location= ebx
| | | 109 end eval block type= 304 status= 401 location= ecx
| | | 110 begin eval block
| | | | 112 begin eval block
| | | | 120 end eval block type= 304 status= 401 location= ecx
| | | 125 end eval block type= 304 status= 401 location= ebx
| | 130 end call sequence args= 2
| 132 end call block free= 1 bound= 0
| 132 begin call block
| 135 end call block free= 0 bound= 0
135 end program block free= 0 bound= 0
```

Figure 3: Block structure of append