Project Description

We propose to develop the theoretical and engineering basis for the *trustless dissemination of software.* We seek to develop the means to distribute and execute software among literally thousands of networked computers without compromising their integrity, while minimizing the need for trust among participants and maximizing the usage of their collective computational resources. To make this possible, we propose to undertake a comprehensive investigation into the use of *certifying compilers* to produce efficient machine code that is equipped with a checkable certificate of compliance with the security, integrity, and privacy requirements necessary for its safe execution on unknown computers. We propose not only to develop the enabling technology, but also to build a demonstration system that allows developers to deploy, rapidly and reliably, applications that make use of the idle computing and storage resources of a network of computers, perhaps spanning the entire Internet.

The general concept of making productive use of idle computational resources has been around for decades. With the advent of the Internet, the sum total of all idle resources is thought to be many times greater than the fastest supercomputers. Therefore, the prospect of harnessing this power has attracted an increasing number of researchers, and in recent years some substantial progress has been made. Many of the early successful applications have been devoted to solving basic problems in number theory and cryptography, such as computing the digits of Pi [3, 44], computing the factors of large numbers [53], and finding large primes [15]. In just the last three years, however, the range of problems being attacked has been expanding rapidly, to include global climate modeling [1], protein folding [42], AIDS drug research [25], the search for extraterrestrial life [57], video animation [21], cancer research [43], and more.

Participation by computer users has also been increasing. For example, in 1997 the Search for Extraterrestrial Life project created SETI@Home, in the hopes that thousands of computer owners might volunteer their excess CPU cycles. The response has been extremely enthusiastic; in May of 2000 the SETI@Home project reported that over two million users were actively running their software. This success has sparked a large number of related developments, leading most recently to the formation of the Global Grid Forum [16], which is a consortium of researchers and developers intended to foster the development of a world-wide distributed computing fabric. Indeed, interest in this mode of computing has reached the point where several commercial enterprises have launched with business models predicated on the ability to harness the Internet's idle computing cycles [11, 12, 13, 43]. This all seems to happen at an opportune time, as developer demand for large-scale computing resources is growing, exemplified not only by SETI@Home, but also by such projects as the National Virtual Observatory [32], which plans to build a massive database of astronomical information and requires large-scale parallel computation to achieve its goals.

While some researchers and commercial enterprises have successfully used the Internet as a massive computer, significant technical hurdles prevent the full benefits of Internet-scale distributed computing to be realized. One set of fundamental problems lies in the nature of distributed computing itself, because it is often extremely difficult or even impossible to divide a large computation into many small pieces in a way that avoids large communication overheads. Interprocess communication is particularly problematic in Internet-scale computing, since many of the volunteering hosts might have slow or occasional links to the Internet (for example, some hosts might connect once a day by modem), and any application that depends on rapid and reliable communication between hosts is therefore not likely to work well.

We do *not* propose to develop new algorithmic techniques for building distributed applications, but rather to investigate the means by which a distributed computing fabric might be provided.

Specifically, we propose to investigate the problem of *how the components of a distributed application may be disseminated to as many hosts as possible and with the greatest exploitation of their resources.* In our view a fundamental technical obstacle to achieving this goal is how to establish an appropriate *trust relationship* between an application developer and the owners of the host computers. This trust relationship is crucial for several reasons. Firstly, host owners need to know before any software is installed that their safety and security requirements will be respected by any applications that are to be hosted on their computers. Secondly, host owners furthermore require a measure of protection against invasion of privacy, so that any uses of personal information can be carefully controlled. Finally, developers must be able to modify and upgrade their applications freely, and as a practical matter, it is important that installing upgrades causes minimal risk and inconvenience for the host owners.[1] (See the SETI@Home web page [54] for a glimpse into the inconveniences of performing upgrades.)

Establishing these trust relationships is especially important for exploiting computing resources on the network. Since users derive little or no direct benefit from the application software installed on their computers, they may be expected to be especially sensitive to the reliability, security, and maintainability of the software. Standing in the way of the establishment of the necessary trust relationships is the fact that the Internet environment simply provides no justification for such trust. Malicious application developers abound, and even benign developers are still often unable to produce safe, reliable software. When an application is run for the host's benefit (as is ordinarily the case), the host's owner is typically willing to assume the risks of unreliable software. But if it is to be run for the benefit of others, the host owners should and will demand strong privacy and security guarantees. In other words, for the purposes of exploiting its idle computational power, the Internet is essentially a *trustless* environment. The fundamental problem, then, is how to generate the necessary trust in a trustless environment.

Today, trust is a matter of faith. Furthermore, in many cases, host owners are forced to reexamine their trust each and every time the developer makes a software upgrade. This is because the host owner is often asked to go through the hassle of downloading and installing the upgraded software explicitly. In other cases, an organization, essentially acting as a "cycle broker" for the developers and as a "quality assurance team" for the host owners, gives a statement of assurance, the value of which is based on the organization's reputation. Sometimes this assurance is buttressed by the use of Java [22] or other technologies, which help to convince the host owners that the assurances can actually be at least partially enforced.

We envision a new paradigm in which *developers freely disseminate their software to willing hosts*, and where *trust is established via rigorous, mathematical proof of the security and privacy properties of software.* This paradigm would operate in concert with the already established economic and social incentives given to owners to make their excess resources available. In a nutshell, our vision is that the developers should gain control over the installation of their software, and the host owners should be relieved of the burden of having to do it themselves and of the worry over whether the software will follow the rules.

[1] There is an additional trust requirement, which is that developers need to know that their application software will not be modified or tampered with in any way that will unknowingly corrupt the results of execution. This problem is typically dealt with in practice by arranging for multiple hosts to work on the same subproblem and then comparing their results. We plan to use the same approach to this so-called *agent integrity problem*, and hence our proposed research will focus on *host integrity.*

A Framework for Trustless Software Dissemination

It is our contention that realizing this vision of Internet-scale computing requires the establishment of appropriate trust relationships. We believe that the following ideas will be essential components of an enabling technology for establishing trust in trustless environments:

1. The precise formal expression and enforcement, by the host computer, of the *safety*, *resource-bound*, and *access control* properties it requires of all applications programs.

2. The developer's use of *certifying compilers* to generate optimized object code equipped with a mechanically checkable certificate of compliance with host's safety and security properties.

At the core of these ideas is the concept of *certified code*, in the forms of *proof-carrying code* [33, 36] and *typed assembly language* [29, 27]. Certified code is code that comes with additional information—essentially a kind of certificate—that allows a host computer to verify, quickly and reliably, the code's safety and security properties. This is in contrast to digital signatures and fingerprints, which are limited to verification of extrinsic properties such as the code's author or point of origin, and in some cases require communication with trusted third parties. It is the intrinsic nature of the certificates that permits trust to be established in trustless environments such as the Internet. Because certified code can be automatically verified for what it will and won't do when executed, it matters much less who wrote it or how it was created or transmitted.

Two important realizations of certified code are proof-carrying code (PCC) and typed assembly language (TAL). In both instances, the code is written in a machine or assembly language that can be highly optimized using conventional compilation techniques. The use of certified machine-level code also means that PCC and TAL programs can access low-level machine resources, and can even be run in kernel mode [33]. This is in contrast to methods based on abstract machines, such as the Java Virtual Machine (JVM) [22], which rely on high-level instructions that are either interpreted to ensure well-behavior, or are translated using a complex (and possibly buggy) just-in-time compiler. Conventional operating system protection techniques suffer from a similar drawback, but at an even coarser level of granularity. Operating systems, by their very nature, provide only very general protection mechanisms, such as address spaces, that are enforced at run-time by a combination of hardware and software checks. Moreover, in practice neither operating systems nor virtual machines may be tailored to specific tasks.

We have been deeply involved in the invention and development of both PCC and TAL in our prior research, and have developed them into richly expressive and highly practical technologies [8, 41]. At present, PCC and TAL provide *type safety assurances* for object code that ensure that the program is confined to its own data and instruction space, and that guarantee that data structures and the run-time stack are not corrupted during execution.

For certified object code to be useful to application developers, there must also be a convenient, flexible means for producing it. Since it is prohibitively difficult to analyze machine code and to determine whether it is safe, much less to produce evidence of its safety, we propose instead to generate safety certificates *automatically* using *certifying compilers* [30, 38, 37]. A certifying compiler determines that a program is safe by examination of the source code (which is invariably simpler than the ultimate executable), and then uses that information together with its knowledge of its own compilation process to produce security evidence for its output executable. This shifts the burden of proof from the code recipient, who is offering computing resources, to the code producer, who wishes to exploit them.

Current certifying compilers are Popcorn [26], Touchstone [35] and SpecialJ [8]. Popcorn and Touchstone are certifying compilers for safe, C-like, low-level languages that provide little support for modularity but are fairly lightweight. SpecialJ is a compiler for full Java that can compile and certify the safety of applications programs as complex as the Hotjava browser and the StarOffice application suite with acceptable overhead in certificate size. We expect the source code of SpecialJ to be available from a commercial source this year. All of these certify type safety and control-flow safety, which are basic prerequisites for all other more complex policies.

Present technology is inadequate to permit trustless dissemination of software in a safe, reliable, efficient, and secure manner. In order to address the demands of such an architecture, we propose to develop the underlying science, testbed implementations, and prototype applications. This requires the synergistic development of these technologies:

- formal policies that guarantee not only safety, but also resource-bound and access control properties of certified code;
- programming language mechanisms that can effectively verify the necessary properties at the source level; and
- certifying compilers that can propagate such source properties and evidence for their validity to low-level code.

As a tangible realization of the basic technology, we propose to build a demonstration system encompassing the necessary basic infrastructure and some demonstration applications. We envision the following experimental framework:

1. A host owner installs a *steward* on his or her machine that runs as a daemon and listens to incoming installation or upgrade requests. During the steward's installation the host owner chooses a *host policy* specifying the security parameters, initially from a fixed menu of choices. This could allocate resources and impose certain access control and authentication requirements. Internally, the host policy is translated into proof- or type-checking obligations that are sufficient to ensure conformance with the host's requirements.
2. An application developer requests resources on various machines by posting software or software upgrades. The posted software is certified with respect to a *developer policy* stating those security and safety assurances that are offered by the developer. Hosts that have available computing resources will consider the request by examining the application binary, which is equipped with a certificate of compliance with the developer's policy. If the application can be proved to conform to the host policy, the binary is accepted and executed (without intervention by the user) on the host machine, with results reported back to the developer via the steward.
3. It will be necessary, especially in early stages of development, to update the steward itself. For this we envision equipping the steward with a "hot swap" capability based on conventional authentication techniques that allows new versions of the steward to be installed without explicit intervention by the host.

This architecture provides the basic infrastructure to achieve trustless software dissemination based on certified object code. The combination of code certification and proof- or type-checking provides a practical framework for the host to state and verify the basic trust relationships required for distributed applications. Whether this framework is useful depends crucially on the policies that can be succinctly expressed and enforced within this overall framework. (We will discuss various possibilities in this direction below.) It should be stressed, however, that the effectiveness of these policies is directly dependent on the integrity of the host system. Faulty or compromised operating environments can negate the guarantees provided by our experimental framework. This limitation is shared by any method for ensuring host integrity, whether it be based on abstract machines, authentication techniques, or the methods we propose to investigate here.

As an initial experimental testbed, we plan to install our dissemination infrastructure on a collection of desktop computers in the School of Computer Science (SCS) at Carnegie Mellon University. This will allow us to exploit the pre-existing culture of deployment of experimental software within SCS, as well as permit early experiments on a controllable and manageable scale. We also envision extending our infrastructure to include hosts at other universities with whom we have established working relationships.

The remainder of this document presents the details of our proposed research project. We organize this presentation into sections, with each section giving an overview of a specific major subproblem, its relationship to the overall research goal, and our plans for addressing it. These major subproblems are as follows: the development of resource-bound and access-control policies and enforcement mechanisms, the design of programming languages for application development, the design and development of certifying compilers, and the use of logical frameworks for efficient proof representation. We conclude the proposal with a brief discussion of our overall research plan and our approach to disseminating our software and research results.

Resource-Bound Policies

A key advantage of both the PCC and TAL approaches to code safety is that they exploit our current understanding of formal type systems. This advantage provides us with substantial leverage, both in our ability to define and reason precisely about code safety properties, and in our ability to obtain well-engineered implementations of safety enforcement mechanisms. Also crucial is the fact that current type systems suffice to express a variety of simple but fundamental properties, such as the type-correctness of memory accesses and the safety of all possible control paths through the program. In a nutshell, type correctness ensures that programs do not violate the integrity of the host system, for example by writing to unallocated memory or by providing illegal arguments to the operating system. This level of safety is extremely important, because real-world experience shows that most system security breaches are caused by programs that either cause buffers to overflow or provide illegal arguments (often intentionally) to system calls.[2]

The practical evidence shows that type safety is essential and has enormous impact on computer security, and indeed it is (approximately) this level of safety that is provided by other systems such as the Java Virtual Machine [22] or Software Fault Isolation [62]. Nevertheless, we believe that more is required in order to achieve the level of trust necessary for people to relinquish full control of their hosts. Specifically, for the purpose of trustless dissemination of software on large networks, we believe that the host systems must be able to constrain the quantity of system resources used by untrusted applications. The resources in question include processor cycles, memory, disk space, and network bandwidth used, as well as the length of time that any such resource—such as the CPU, a block of allocated storage, or network device—can be held before yielding it to another process. We refer to limits on the use of system resources as *resource-bound* properties of applications. To achieve the necessary level of trust, host owners must be able to specify the resource-bound properties they require and have those properties enforced by the steward.

[2]See http://www.cert.org/summaries/CS-2000-04.html for a recent summary of reported security violations. In addition to buffer overflow and illegal arguments, a third major category of security breaches is due to Microsoft Visual Basic scripts. We avoid this category of problems in our proposed system simply by avoiding Visual Basic.

In previous work [34], we conducted some preliminary investigations into one approach to controlling the use of CPU, memory, and network resources. In essence, this approach involved inserting dynamic checks into the untrusted code in order to render it "easy to prove," and then optimizing away the checks while at the same time deriving the (more complicated) proof or typing annotation of the optimized program.

Before describing the application of this idea to resource bounds, let us begin with a simpler but analogous situation. Consider the problem of determining the safety of accesses to the elements of an array. In general it is extremely difficult to prove that an array access is within the bounds of the array. However, there is a way to render it easy to prove, namely by inserting code *just before the array access* that dynamically checks that the array access will be within bounds. Of course, checking every array access in this manner is both naive and inefficient,[3] and hence most good compilers try to minimize such checks. The point is that the compiler (if it is correct) performs optimizations for a sound reason, so if the compiler is able to perform such an optimization for an array access, then *it is also able to generate a proof or a typing annotation that makes it easy for the host system to conclude that the array access is within bounds.*

Now, in the case of resource bounds, we can approach the problem in a similar way. Consider, for example, the problem of limiting the use of the CPU by specifying an upper bound on the total number of instructions executed. A naive approach is simply to insert code before every instruction in the program that increments a global counter and checks whether the counter exceeds the specified upper bound, and then aborting execution of the program if this should ever happen. For code such as this, it is not at all difficult to construct a proof that the program runs within the specified bounds. A compiler can then attempt to optimize away such checks and, as with array bound optimization, when the compiler is able to do so, it can also construct a proof justifying the optimization. Performing such optimization for straight-line code is trivial, and as we have shown in previous work, simple loops are also relatively easy to optimize into single checks that are executed once per iteration or, in some cases, hoisted out of the loop altogether. Much harder is the problem of optimizing the instruction-counting checks in programs that contain nested loops, though there seem to be similarities, at least on the surface, to compiler optimizations such as software pipelining [6, 49] and loop transformations for improving cache performance [65].

Note that the combination we propose of static verification (using proofs) and dynamic verification (using run-time checks) is necessarily more efficient than dynamic verification alone, and likely much more efficient. This is for two reasons. Firstly, there is the practical matter that a purely dynamic regime usually requires a context switch (associated with either a system call or an interrupt) before any dynamic check can be performed, whereas in our approach all dynamic checks are performed by the application itself. More fundamentally, however, a purely dynamic regime must perform every single check, while in our approach many checks can be optimized away.

A variety of other resource-bound properties can be certified in a similar way. For example, similar techniques can be used to limit the amount of memory allocated, disk space used or the number of bytes written to the network. For another example, it is also possible to use this approach to deny access to the network device unless a specified minimum number of instructions have been executed since the last network access [34], thereby limiting bandwidth usage.

Despite the potential difficulties in optimizing away the dynamic resource checks, particularly in programs that exhibit complex control flow, the underlying approach in its simplest form appears to be easy to understand and straightforward to implement. Furthermore, we have found that it is possible to capture this approach in a *resource bound calculus*, and use this to create an extension

[3]Despite the naivete of checking every array access, some systems for enforcing code safety, such as the Java Virtual Machine, do in fact check every array access. This is further evidence of the importance of even the most basic safety properties.

to the TAL type system [9]. From this point, it seems clear that recent work on typechecking for languages with restricted forms of dependent types, such as DML [69] and DTAL [67], will allow this calculus to capture some forms of optimizations. This suggests a clear plan for our proposed research in this area. We begin with the simple approach that we have taken already, suitably extended for the steward application. This will provide a basic level of support for resource bounds, allow larger-scale experience to be gained, and provide a concrete target for formalization of the breadth of resource bounds required by donors. This experience will then provide us with guidance on the shortcomings of the approach and the need for optimizations that will likely require significant development, along the lines of major control-flow transformations such as software pipelining. It is also possible that the programmer could assist in the optimization of resource checks by supplying resource-usage annotations in the original source program [9]. Finally, we plan to study the general question of more complex specification and verification of resource-bound properties in parallel with the experimental activities in this area.

Access Control Policies

In its usual form, type safety protects sensitive operations by denying access to such operations entirely. For example, system calls can only be made when used in a manner consistent with their specification in the type system (thereby ensuring that appropriate arguments are used); by simply omitting a system call from the type system, that system call is made completely inaccessible. Hence, it is easy to ensure that verified applications cannot (say) format the host's hard disk. However, complete inaccessibility is inappropriate for some sensitive operations. For such operations, we must provide access, but provide it in a controlled manner to avoid damaging the host's owner's interests.

For the most part, the controlled access we require can be provided using ordinary type safety techniques. For example, if applications are permitted to create files for temporary storage, then applications will probably also be permitted to delete those files so as to clean up after themselves (indeed, a resource-bound policy will probably require them to do so). Nevertheless, the host's owner certainly will not desire to give applications the unfettered ability to delete files. A natural policy is to allow an application to delete only files it created. This can be enforced using ordinary type safety by taking advantage of type abstraction [50]: The type system specifies that the delete file system call requires a *handle* to the file to be deleted, and the only way to obtain a handle is using the system call to create a file. If the type of file handles is made abstract, then the application cannot forge a handle, and consequently applications are limited to deleting only files they can legitimately obtain handles to, those it created.

Another access control issue that is important in many mobile code applications, but that does not arise for our proposed system, is the so-called "luring" attack [51]. In luring attacks, trusted code calls untrusted code, providing it a callback (back into trusted code), which the untrusted code abuses to perform an unsafe operation.[4] In a sense, the blame for such attacks can be assigned to the original trusted code for providing an abusable callback to untrusted code, but it can be very difficult in practice to program in a manner immune to such attacks, which has led developers to desire additional protection against such attacks. Java implementations typically do so using a technique called stack inspection [64] in which the system, before performing a possibly unsafe operation, checks that no untrusted stack frames lie below a trusted stack frame that authorized the operation. This can be costly, and a variety of alternatives have been proposed [64, 55]. In

[4]The idea behind the term is that the untrusted code "lures" some trusted code into providing the abusable callback.

any case, luring attacks do not arise in our proposed system. Luring attacks can occur only when trusted code calls untrusted code, but in our proposed system the only trusted code that will ever call untrusted code is the steward itself.

Type safety provides a notion of unforgeable capabilities—in the above example the capability to delete a file. In some circumstances it is useful to extend this facility to *revoke* capabilities, for example, a host owner might wish to grant the application the capability to write to the network only when network load is light. Revocable capabilities can be supported using standard dynamic means [66], in which capabilities are checked for validity (*i.e.*, for not having been revoked) at run time before operations are performed. For most applications, dynamic validity checking is satisfactory, but if run-time checking were to be too costly (such as with a memory-mapped network interface, perhaps), validity checking can also be performed statically using a type system of capabilities [63, 59]. The same sort of type system can also enforce *cleanup* properties [63]; for example, it can ensure that applications delete any files they open.

A final issue related to access control is that of *information flow* properties. An information flow property dictates would dictate that, although an application may be permitted access to some piece of information, it is not permitted to propagate that information to any agent not permitted access, such as the application's developer. Information flow properties are very useful in settings of cooperative computation, but have not the same degree of importance here, since applications have little legitimate reason to require access to the host's data at all. Nevertheless, a few legitimate reasons do arise; for example, an application may wish to consult the host's system configuration in order to optimize its own behavior or install software updates, but the host's owner may not wish to allow the configuration to be reported to the outside world. The current work in this area [20, 31, 61, 71] has made some strides toward enforcing information flow properties, but none yet seem flexible enough for practical use in large applications. For our initial testbed and applications, we expect to enforce information flow using access control; thus, applications will simply not be permitted to consult the system configuration.

Programming Language Design

Our approach to trustless software dissemination is based on code certification. At present the most practical way to certify that a program complies with a stated policy is to use a *certifying compiler* that equips its object code with a checkable certificate of compliance with source code safety and security policies. For this to be feasible, the compiler relies on a combination of theorem proving techniques and the insertion of run-time checks in the object code. In essence a theorem prover is used to determine whether a run-time check can be eliminated, and the object code is augmented with a formal proof of this fact.

While this strategy has proved effective for relatively simple source-level safety policies [36, 29], it is a significant open question whether it can be made to work for properties such as resource bounds without incurring unacceptable performance penalties. An important part of our proposed research is to explore the design of new programming language constructs that support certification of distributed applications software. We propose to address this problem in two stages. Early on we intend to focus on *low-level languages* that support the construction of small, efficient programs that make few demands on the ambient computing context. These will work best for compute-intensive applications, with little or no communication or storage requirements. Here the emphasis will be on support for resource-bound and access-control certification. Later in the project we expect to shift attention to *high-level languages* that support modular programming, richer forms of policy specification, and mechanisms for mobile computing. Here the emphasis will be on providing linguistic support for building applications that take full advantage of a rich, distributed computing environment.

In the early stages of the project we intend to build applications using extensions of either Popcorn [28] or Touchstone [36, 38], both of which compile safe C-like languages. These are well-developed certifying compilers that are suitable for experimenting with new certification techniques, and which support development of relatively simple, compute-intensive applications.

Safety in this context refers to memory safety (no illegal memory accesses), control-flow safety (no illegal jumps), and type safety (no violation of data integrity). These safety properties are mutually dependent and form the basic requirements for any certification of higher-level properties. They are guaranteed by a combination of source language restrictions and run-time checks that are eliminated when provably redundant.

Since program properties are much easier to specify than to discover, we plan to extend the source language so that resource-bound properties and restrictions can be expressed by the programmer within the program text. Concretely, these will most likely take the form of logical assertions (pre-conditions, post-conditions, and other invariants). However, unlike basic type safety, fine-grained bounds on resource consumption are not intrinsic to the source program, but require a combination of information from the source program and the compiled code. By keeping the operational model of the source language simple, for example, by avoiding garbage collection, we can keep this gap small.

In the later stages of the project we intend to investigate the design of more sophisticated high-level languages with richer type systems (or other specification formalisms). For example, our previous research on *refinement types* (both in the form of sorts for data structures [14, 10] or dependent refinements [69, 70, 67, 68]) can be expected to play an important role. Refinement types provide a flexible and expressive means of stating and verifying program invariants such as the range of values of an integer expression or the assertion that an object reference is non-null. Our previous research focuses on the use of refinement types as a software engineering tool. We intend to investigate the use of refinement types in a certifying compiler to enable efficient certification of safety and security policies needed for trustless distributed computing.

Besides our experience with lower-level languages from the first phase of the project, work on type-based compilation for ML [58] and certifying compilation for Java in the SpecialJ compiler [8] will provide a point of departure for the second phase.

In the longer term it will also be important to provide language support for building distributed applications. This will involve not only establishing the assurances required by hosts, but also the mechanisms for propagating tasks on the network, communicating among them, and collecting their results. Here we envision the need for *network-sensitive* languages such as Cardelli and Gordon's *mobile ambients* [4], which make explicit concepts such as movement into and out of security domains. Current, experimental type systems for ambients [5] are able to express properties about the communication and mobility of ambients. It would be interesting to consider how to build a certifying compiler for mobile ambients that would extend these properties to distributed object code.

Certifying Compilation

The third area of focus for our proposed research is the design and implementation of *certifying compilers* for high-level source languages, as described above. The certifying compilers we propose to develop are intended to be the primary tools used by application developers. They will provide the means for automatically generating certified target programs from high-level source programs. In order to describe our proposed work in this area, we begin by describing our prior research accomplishments, and then discuss the open research questions that are directly relevant to the goals of this proposed project.

We have considerable prior experience, having established this research area in our development of the concepts of *type-directed compilation* [30, 58] and *typed intermediate languages* [18]. Currently, we have an ongoing large-scale implementation effort to develop a type-directed compiler for Standard ML, called TILT. These concepts later inspired the Touchstone [37] and Popcorn [28, 29] certifying compilers, which automatically generate certified native-code binaries. We have also been involved in the development of SpecialJ [8], a commercially developed high-performance compiler for Java that is scheduled to become an open-source resource later this year. It generates optimized PCC binaries for the Intel x86 architecture and is in a late stage of its development. Also related is our development of certifying theorem proving technology [40, 7], which is used in both the Touchstone and SpecialJ compilers.

In this prior work, we have focused first and foremost on developing the theoretical foundations of certifying compiler technology, and then on investigating its applicability to programming languages of realistic size and complexity. The TILT compiler effort, for example, is in part an investigation into the suitability of the Harper-Stone type-theoretic interpretation of Standard ML [19] as the basis for the design of a typed-intermediate language for an ML compiler.

A second major focus of our prior work has been on the generation of very high-performance certified code. In principle, certifying compilers ought to be able to generate highly optimized native-code binaries whose certificates ensure their safety. To this end, the Touchstone and Popcorn compilers have devoted considerable effort into techniques for automatically certifying optimized machine code, with encouraging early results [38].

Despite our extensive prior work in this area, there are several key aspects of the problem of trustless software dissemination that are not addressed by this prior work. Specifically, relatively little has been done, to-date, on the propagation of resource bounds and access controls in a certifying compiler, and the development of corresponding enforcement mechanisms for the steward. Furthermore, the problem of providing a flexible and possibly extensible framework for security policies in a certifying compiler has been largely ignored. In the context of Internet-scale computing, however, it seems that such flexibility is essential, as it would allow some level of customization of security policies by both host owners and developers. Finally, there are practical system-engineering issues, for example having to do with how to determine and maintain the consistency of compilers and stewards in a system where thousands of stewards and compilers might be deployed.

Besides these open research questions, which are largely specific to the goals of this proposal, there are also several substantial open research problems in the underlying theory and engineering of certifying compilers, and these are also directly relevant to the research goals of this proposal.

One major set of questions concerns the practicality of the certificates. In the case of TAL-based certificates, there are questions regarding the size of the typing annotations, the time required to perform typechecking, and the complexity of the typechecker. In the case of PCC-based certificates, there are corresponding questions regarding the size of join-point annotations and the time and space required for proof-checking. The TAL typing annotations and PCC join-point annotations can become extremely large, in principle reaching exponential proportions. While the problem of optimally controlling this growth is not tractable, we have some experience in appropriate heuristics, and find that there is substantial room for research on this problem.

Another major question is the interaction between the steward and the compiler. The compiler certifies the object code to comply with a developer policy stating safety properties of the program. The steward requires compliance with the host policy, which states the conditions under which the host is willing to execute applications. How are these to be reconciled, and how might they evolve over time? In the framework we envision it is the obligation of the steward to check that the developer policy logically entails the host policy. In principle this could involve arbitrary deduction, which would, of course, be impractical. In the early stages of the project we intend to limit this entailment by limiting the specification of host policies to a few simple conditions that can be easily checked against the developer policy. In the longer term it is a significant research problem to develop more flexible methods for ensuring compliance with a richer variety of host policies.

Logical Frameworks

The representation and verification of safety proofs plays a critical role in any architecture for distributed computing based on proof-carrying code. Consequently, proof-carrying code has become a major application area for *logical frameworks*, that is, meta-languages for the representation of deductive systems. In a logical framework language, the basic elements of proof systems, including the syntax of predicates, the axioms and inference rules for constructing proofs, and even aspects of the meta-theory, can all be represented and manipulated as objects. The logical framework LF [17] is particularly well suited for such applications, since proofs can be represented as first-class objects and, importantly for the PCC application, checking the validity of a proof reduces to typechecking its LF representation.

In our previous investigations, we discovered that pure LF representations of proofs, while fast and easy to validate by using a simple LF typechecker, were unreasonably large. In practice, some proofs would be thousands of times larger than the programs whose safety they were proving. One of our earliest advancements to overcome this problem was based on previous developments in logical frameworks and logic programming [23]. This led to techniques for proof compression and efficient verification of a useful fragment of LF [39], called LF_i, and formed the basis of the first practical implementations with manageable proof sizes and fast validation. In practice, the LF_i representation of proofs led to proof sizes roughly the same size as the code and near-linear-time validation, even for Java-based applications of realistic size and complexity [8].

Although LF_i made it possible to claim a certain level of practicality for PCC, a 100% overhead is still significant, especially when compared to technologies such as cryptographic signatures. In later work, Necula and Rahul developed the oracle-based representation of proofs for PCC [41]. In this approach, the validation is performed by a logic-programming interpreter for a fragment of LF, in which nondeterministic choice and higher-order unification are guided by "oracle bits" provided with the code. Essentially, the oracle bits thus represent the proof. This approach has been implemented in the SpecialJ system, with extremely good results. Typically, for realistic Java applications, the number of oracle bits is only a small fraction (usually less than 5%) of the code size.

Despite the major progress that we have made in proof representation and validation, there are significant new challenges posed by the problem of trustless software dissemination. Resource-bound and access-control certification will require more complex arithmetic reasoning than is required in present safety policies. (One intuition for this is to observe that array accesses are based primarily on addition of small non-negative integers and multiplication by 4 or 8, whereas resource bounds may involve reasoning about arbitrary nested loop iterations.) In this case, it is possible that the oracle-string representation will turn out to be brittle, with small changes in a signature having a major impact on the size of the oracle string. Moreover, oracle strings are not compositional, which means that adding two signatures, or adding a single new declaration to a signature requires that the oracle strings be completely redone. In the context of trustless software dissemination, this may lead to significant practical difficulties in the maintenance and upgrade of the disseminated software.

Indeed, the pure LF-based encodings have many nice properties. Besides being fast and easy to validate, they are compositional and therefore locally checkable (in contrast to oracle strings, which are only globally checkable). While this seems to come at a high price in terms of size of the representation, we see significant possibilities for developing new compression techniques. [5]

We therefore plan to investigate techniques for highly compact LF representations of PCC proofs. We believe the basic architecture of higher-order constraint logic programming [24] and certifying decision procedures [40] in the context of LF is sufficiently robust to permit such richer domains, but significant further research will be required both in theoretical foundations and implementation technology. We propose to develop these foundations based on some initial results [60, 45] and integrate them into the Twelf system [46] which is already used for projects in proof-carrying code and certifying decision procedures at Princeton [2], Yale, and Stanford [56], as well as at CMU. One approach that seems possible is to investigate the use of signature transformation and other meta-theoretic tools. Another approach is to prove the decidability of certain predicates by LF termination arguments and then reconstruct them during the validation process. Ultimately, we plan to develop a range of approaches that allows us to "tune" the tradeoff between size of representation, the time required to validate proofs, and the ability to compose and maintain them.

Since the verification of certificates is at the heart of the trusted computing base in our proposed architecture, there is also a strong need for the formal analysis of safety and resource bound policies and their specifications. These may have the form of a logic (as in proof-carrying code) or a type system (as in typed assembly language). We propose to develop techniques for reasoning about properties of such specifications. Initial steps in this direction have been taken [47, 52], but further fundamental advances are required, for example, to integrate constraint domains into meta-theoretic reasoning.

Research Plan and Dissemination

There are two primary means for dissemination of our research results: publication of scholarly papers and distribution of software artifacts. The construction, maintenance and documentation of our implementation components requires a major engineering effort. At the beginning of the project we plan to implement a primitive steward that can receive work requests, verify the type safety of binaries in the TAL language, and return results. This establishes the basic framework for our project, which can be roughly divided into three phases to be accomplished over the five-year grant period.

The first phase can be characterized as group internal development and experimentation. We plan to build several demonstration applications such as a program for prime factorization and a distributed theorem prover. We propose to evaluate the costs of certification and checking and the practicality of the initial programming model while at the same time advancing the certified code framework. This includes work towards richer policies (for example, access control and resource bounds), more flexible policy specification and verification, and designing and implementing a certifying compiler for more expressive policies.

The second phase includes the local deployment of a prototype steward at CMU and by close colleagues elsewhere and the joint development of distributed applications with local domain experts such as distributed ray tracing or other algorithms from the graphics and scientific computation domains. At the same time we will evolve the theoretical foundations, the certifying compiler, and

[5] The term "compression" is not quite right, since our intention is to achieve small proof representations that are directly checkable, thereby avoiding the complexity and overhead of an explicit decompression step in the host computer.

the steward based on the results from early experimentation. In the realm of language design we plan to provide language support for distributed computation and incrementally enrich the source language to satisfy the demands of more complex applications.

The third phase will be focused on the work necessary to permit world-wide deployment of the stewards, additional support for application developers, and continued evolution of certification and verification techniques and their implementations. Furthermore, in order to allow formal and informal evaluation and foster trust and scientific inquiry, the documentation will have to be significantly more rigorous than is typical in both the research and commercial domains. It will include, among other items, a formal specification of resource bound certificates and their verification.

Broader Impact

The proposed work draws upon mathematically deep, theoretical foundations such as type theory and logic. Through our significant experience with the engineering of practical compilers, theorem provers, and run-time systems, these can be put to use in order to exploit vast untapped resources in our national and international computing infrastructures.

There are a variety of incentive models for donors of computing resources. The simplest is a policy decision within an organization such as a business or a university which has a vested interest in exploiting its own resources to the fullest extent possible. Currently, security concerns and a lack of trust are indeed a major obstacle to such internal adoptions of distributed computing models and can be overcome with our proposed research.

Another is a general desire by many to further science and the welfare of society. Currently, this is the model under which SETI@Home operates successfully on a world-wide basis. The scope of such efforts that rely on scientific computations amenable to distributed computing methods could be extended significantly if one could make confident assurances as to the safety, reliability, and unobtrusiveness of such software.

Recently, there have also been a number of business models in which services such as free e-mail or Internet access are bartered for computing cycles of idle machines [48]. Such efforts are generally viewed with justifiable distrust by consumers, and the risks are becoming well publicized in the popular press [48]. We believe that our proposed work can make a difference in this arena as well. The commercial success of the Java model for mobile code is at least a partial testimony to the increased attention to safety.

Finally, it is clear from the distributed computing efforts mentioned in the introduction, that there are many computational problems of scientific and even commercial interest that are amenable to distributed algorithms and would greatly benefit from the infrastructure we are proposing to provide. Programming should be significantly easier, resulting in programs safe and significantly more reliable than possible with present technologies.

Ultimately, the proposed research, in addition to extending the field of programming languages and especially the applications of formal semantics and type theory, has the potential to create a software infrastructure that can serve the needs of major, large-scale computing efforts.

Education and Training

Besides the direct impact in the domain of distributed computing, we believe our work will significantly advance our understanding of programming languages and the application of formal semantics to real-world problems. As a project that will support six graduate students and a post-doctoral researcher, it provides an ideal laboratory for education in computer science.

We also plan to offer three summer and part-time jobs for enterprising undergraduates at Carnegie Mellon University and other institutions that provide a sufficiently broad background. In this way the project will also directly contribute to undergraduate training.

We also plan to collaborate with clients who are interested in writing software and executing it in a distributed fashion. This requires additional educational effort for students in other disciplines is part of the current proposed work.

References

[1] Myles Allen. Do-it-yourself climate prediction. *Nature*, 401(642), 1999.

[2] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In Thomas Reps, editor, *Conference Record of the 27th Annual Symposium on Principles of Programming Languages (POPL'00)*, pages 243–253, Boston, Massachusetts, January 2000. ACM Press.

[3] Fabrice Bellard. Fabrice Bellard's PI page. URL: `http://www-stud.enst.fr/ bellard/pi`, August 1997.

[4] Luca Cardelli. Mobility and security. In Friedrich L. Bauer and Ralf Steinbrggen, editors, *Proceedings of the NATO Advanced Study Institute on Foundations of Secure Computation*, NATO Science Series, pages 3–37, Marktoberdorf, Germany, 1999.

[5] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile agents. In Alessandro Fantechi Paolo Ciancarini and Roberto Gorrieri, editors, *Formal Methods for Open Object-Based Distributed Systems*, Florence, Italy, February 1999. IFIP TC6/WG6.1.

[6] A.E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120b/FPS-164 family. *IEEE Computer*, 14(3):18–27, 1981.

[7] Christopher Colby, Peter Lee, and George C. Necula. A proof-carrying code architecture for Java. In E.A. Emerson and A.P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, pages 557–560, Chicago, Illinois, July 2000. Springer-Verlag LNCS 1855.

[8] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 95–107, Vancouver, Canada, June 2000. ACM Press.

[9] Karl Crary and Stephanie Weirich. Resource bound certification. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 184–198, Boston, January 2000.

[10] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In P. Wadler, editor, *Proceedings of the Fifth International Conference on Functional Programming (ICFP'00)*, pages 198–208, Montreal, Canada, September 2000. ACM Press.

[11] DCypher.Net. URL: `http://dcypher.net`, November 1999.

[12] Distributed Science, Inc. URL: `http://www.processtree.com`, 1999.

[13] Entropia. URL: `http://www.entropia.com`, 1997.

[14] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*, pages 268–277. ACM Press, June 1991.

[15] GIMPS. Great Internet Mersenne Primes Search. URL: `http://www.mersenne.org`, January 1996.

[16] The Global Grid Forum. URL: `http://www.gridforum.org`.

[17] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[18] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[19] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[20] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 365–377, San Diego, California, January 1998. ACM Press.

[21] Johannes Hubert. Internet Movie Project. URL: `http://www.imp.org`, 1998.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[23] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[24] Spiro Michaylov and Frank Pfenning. Higher-order logic programming as constraint logic programming. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 221–229, Newport, Rhode Island, April 1993. Brown University.

[25] Garrett M. Morris. FightAIDS@Home. URL: `http://www.fightaidsathome.org`, September 2000.

[26] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[27] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 2000. To appear. An earlier version appeared in the 1998 Workshop on Types in Compilation, volume 1473 of Lecture Notes in Computer Science.

[28] Greg Morrisett et al. URL: `http://www.cs.cornell.edu/talc`.

[29] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[30] John Gregory Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. (Available as Carnegie Mellon University School of Computer Science technical report CMU–CS–95–226.).

[31] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999.

[32] The National Virtual Observatory. URL: `http://www.srl.caltech.edu/nvo/`, October 1999.

[33] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, October 1996.

[34] George Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, October 1997.

[35] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, June 1998.

[36] George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.

[37] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[38] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, June 1998. ACM Press.

[39] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.

[40] George C. Necula and Peter Lee. Proof generation in the Touchstone theorem prover. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, pages 25–44, Pittsburgh, Pennsylvania, June 2000. Springer-Verlag LNAI 1831.

[41] George C. Necula and S.P. Rahul. Oracle-based checking of untrusted software. In Hanne Riis Nielson, editor, *Conference Record of the 28th Annual Symposium on Principles of Programming Languages (POPL'01)*, London, England, January 2001. ACM Press.

[42] Vijay Pande. Folding@home. URL: `http://www.stanford.edu/group/pandegroup/Cosm`, September 2000.

[43] Parabon Computation. URL: `http://www.parabon.com`, June 2000.

[44] Colin Percival. PiHex. URL: `http://www.cecm.sfu.ca/projects/pihex`, February 1999.

[45] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 179–193, Kloster Irsee, Germany, March 1998. Springer-Verlag LNCS 1657.

[46] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[47] Mark Plesko and Frank Pfenning. A formalization of the proof-carrying code architecture in a linear logical framework. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[48] Associated Press. Researchers tapping idle home computers: Distributed computing used for complex tasks. Pittsburgh Post-Gazette, March 19 2001.

[49] B.R. Rau and J. Fisher. Instruction-level parallel processing: History, overview, and perspecctive. *Journal of SuperComputing*, 7(1/2), January 1993.

[50] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress.

[51] Jim Roskind. *Evolving the Security Model for Java from Navigator 2.x to Navigator 3.x.* Netscape Communications Corporation, 1999. URL: `http://developer.netscape.com/docs/technote/security/sectn1.html`.

[52] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems.* PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

[53] RSA Security. DES Challenge III. URL: `http://www.rsasecurity.com/rsalabs/des3`, January 1999.

[54] SETI@Home. URL: `http://setiathome.ssl.berkeley.edu`, November 2000.

[55] Christian Skalka and Scott Smith. Static enforcement of security with types. In *2000 ACM International Conference on Functional Programming*, pages 34–45, Montreal, September 2000.

[56] Aaron Stump and David L. Dill. Generating proofs from a decision procedure. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[57] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on Project Serendip data and 100,000 personal computers. In C.B. Cosmovici, S. Bowyer, and D. Werthimer, editors, *Astronomical and Biochemical Origins and the Search for Life in the Universe, Proceedings of the Fifth International Conference on Bioastronomy*, Editrice Compositori, Bologna, Italy, 1997. ACM Press.

[58] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

[59] Vault: A programming language for reliable systems. URL: `http://research.microsoft.com/vault`.

[60] Roberto Virga. *Higher-Order Rewriting with Dependent Types.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, September 1999. Available as Technical Report CMU-CS-99-167.

[61] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th Symposium on Principles of Programming Languages (POPL'00)*, pages 268–276, Boston, Massachusetts, January 2000. ACM Press.

[62] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, December 1993.

[63] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4), July 2000. An earlier version appeared in the 1999 Symposium on Principles of Programming Languages.

[64] Dan Wallach. *A New Approach to Mobile Code Security.* PhD thesis, Department fo Computer Science, Princeton University, Princeton, New Jersey, January 1999.

[65] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.

[66] W. A. Wulf, R. Levin, and S. P. Harbison. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill, New York, NY, 1981.

[67] Hongwei Xi. Imperative programming with dependent types. In M. Abadi, editor, *Proceedings of the 15th Annual Symposium on Logic in Computer Science (LICS'00)*, pages 375–387, Santa Barbara, California, June 2000. IEEE Computer Society Press.

[68] Hongwei Xi. Dependent types for program termination verification. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, Boston, Massachusetts, June 2001. IEEE Computer Society Press. To appear.

[69] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.

[70] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.

[71] Steve Zdancewic and Andrew C. Myers. Confidentiality and integrity with untrusted hosts. Technical Report 2000-1810, Cornell University, August 2000.