# iProve: A Scalable Technique for Consumer-Verifiable Software Guarantees

Silviu Andrica, Horatiu Jula, and George Candea
*School of Computer and Communication Sciences*
*École Polytechnique Fédérale de Lausanne (EPFL), Switzerland*

## Abstract

*Formally proving complex program properties is still considered impractical for systems with over a million lines of code. We present iProve, an approach that enables guaranteeing useful properties in large Java systems. Desired properties are proven in iProve as a combination of two proofs: one of a complex property applied to a small piece of code—a nucleus—using existing theorem provers, and a proof of a simple property applied to the rest of the code—the program body—using iProve. We show how iProve can be used to guarantee properties such as communication security, deadlock immunity, data privacy, and resource usage bounds in Java programs with millions of lines of code. iProve scales well, requires no access to source code, and allows nuclei to be reused with an unlimited number of systems and to be written in verification-friendly languages.*

## 1 Introduction

Programs we use everyday come with no provable correctness guarantees about their behavior. Instead of competing on provable reliability, software vendors tend to focus on features.

Formal methods offer a rich variety of techniques for proving properties about programs, but formal proofs of non-trivial properties are typically limited to small programs (up to thousands of lines of code, or KLOC) that have only limited interaction with their environment. Proofs entail substantial human effort and, in larger programs, mostly low-level properties have been proven so far, such as functional correctness of linked data structures [18]. However, if we are to provide consumers with software guarantees, we must prove *system-level* properties.

We accept that the only way to leverage current proof techniques for non-trivial software properties is to apply them to small bodies of code. Since real systems are large (e.g., Firefox has over two million lines of code, or MLOC), we endorse a pragmatic divide-and-conquer approach: the property to be proven is split into a complex property and a simple property. The complex property can be proven on only a small part of the code (a *nucleus* < 1 KLOC), while the simple property is proven on the entire code (the *body* > 1 MLOC). A modular combination of the two proofs yields a proof of the complex property for the entire program.

In iProve, a program property is concentrated into a formally verified component, the nucleus, which can be joined to any program. When a nucleus is joined to a program, the latter is automatically modified to pass control, at key points in its execution, to the nucleus, which in turn enforces the property over the entire program. To provide software with guarantees, one must prove the correctness of the nucleus and the correctness of its join to a program body. iProve relies on third-party theorem provers to verify that a nucleus is correct, and on its own join verifier to check the program-nucleus join.

For example, we built a deadlock immunity nucleus based on Dimmunix [8], which modifies a program's thread schedule so as to avoid deadlocks encountered in previous runs. Dimmunix saves fingerprints of discovered deadlocks in a local persistent history file. Prior to lock acquisitions and releases, the program must yield control over to Dimmunix, which decides, based on this local history, whether to delay the lock operation or allow it to proceed. We proved, using Jahob [10], that the Dimmunix nucleus avoids deadlocks if invoked before and after each lock operation. Using iProve, we guarantee deadlock immunity in large systems, like JBoss, ActiveMQ, and Limewire.

With iProve, the cost of building formally verified nuclei is easy to amortize by reusing nuclei across large numbers of programs. For example, Dimmunix can be used with any Java program. Program bodies can be written in widely-used languages (e.g., Java, Scala), while nuclei can be written in languages better suited for proofs (e.g., Haskell, Prolog). Programs are glued to nuclei by the vendor before shipping the software. Nuclei are verified by the consumer prior to installing or running the software. The join verifier runs whenever a program starts executing, to ensure that the join is correct. In essence, iProve's goal is to "discipline" software, without extending its functionality.

*Appears in Proceedings of the Intl. Conf. on Dependable Systems and Networks (DSN), Chicago, IL, June 2010*

The contributions of this paper are: (1) a practical means of provably constraining a program's execution to obey certain properties, and (2) experimental evidence that formal methods can be made practical for large programs.

The paper is organized as follows: After describing iProve's design (§2-§5), we present an implementation for Java programs (§6), we demonstrate the utility of iProve through four case studies (§7), and do a systematic performance evaluation (§8). We close with a discussion (§9), related work (§10) and conclusion (§11).

## 2 iProve Overview

Before describing the iProve components in depth, we provide an overview of the entire iProve system (Figure 1).

There are three distinct parties involved: a system vendor $V_S$ (e.g., Microsoft, RedHat), a nucleus vendor $V_N$ (e.g., a boutique software firm), and a consumer $C$. $V_S$ builds software $S$ and provides it to $C$. $V_N$ produces a nucleus $N$ that is meant to enforce desired property $Q$ in programs (e.g., all communication is always encrypted with 1024-bit RSA). For the nucleus to be effective, it must be joined to system $S$ through instrumentation that is described by so-called join conditions, also provided by $V_N$. These conditions specify $P$, a set of program points in $S$ where $N$ must be invoked if $N$ is to indeed guarantee $Q$ over the program. The nucleus vendor proves that desired property $Q$ holds for any program that is connected to $N$ via instrumentation described by $P$. Informally, $V_N$ proves that $N$ ensures $P \Rightarrow Q$.

Nucleus $N$ is glued to system $S$ using the iProve joiner. The iProve joiner takes as inputs the system $S$, nucleus $N$, and the join conditions $P$, and produces a new version of $S$ that is tied to $N$ as described by $P$. The joiner need not be trusted, since the instrumentation is always checked by the iProve verifier prior to running $S$.

The iProve verifier consists of the nucleus verifier and the join verifier. The nucleus verifier automatically verifies $N$'s correctness ($P \Rightarrow Q$ holds) and the join verifier automatically checks that system $S$ is indeed instrumented as required by the nucleus ($P$ holds). If $P \Rightarrow Q$ and $P$ hold, the desired property $Q$ clearly holds for system $S$. Thus, iProve verifies that the nucleus will indeed enforce its property $Q$ over the entire program.

We envision a usage model in which a system vendor $V_S$ purchases or licenses nuclei from nuclei vendors, and joins the nuclei to a product before shipping it. $V_S$ distributes the sources of the employed nuclei together with the nucleus join conditions; however, the code of $V_S$'s product $S$ need not be distributed. Customers configure their environment to run iProve automatically, to check the correctness of the nucleus and the joining before running $S$. If verification succeeds, customers can rest assured the execution of the program will satisfy the nuclei-enforced properties.

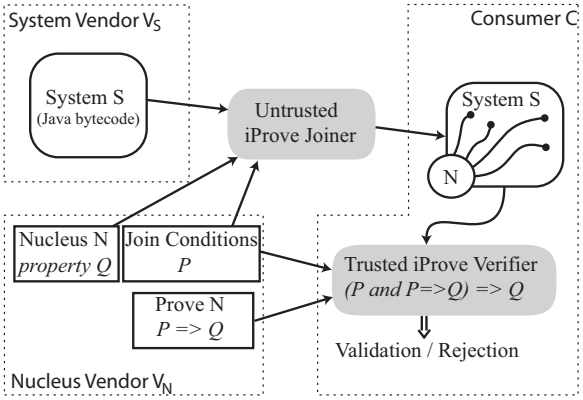



**Figure 1. Overview of the iProve approach.**

Next, we describe how to build a nucleus (§3), how to "glue" it to a program (§4), and how the program/nucleus pair is verified by iProve (§5).

## 3 The Formally Verified Nucleus

A nucleus is a body of code that enforces a desired property, as long as it is correctly connected to a program via specific instrumentation.

iProve draws inspiration from aspect oriented programming (AOP) [9], a programming approach that increases modularity through separation of concerns. In AOP, concerns cut across multiple program abstractions and are joined to the program using an aspect-oriented compiler. By taking an aspect-oriented approach, we separate the concern of provability from that of efficient development: a nucleus is written in a way that is easy to prove, while the program is written in a way that is easy to develop and optimize. This AOP-inspired approach may incur some runtime overhead, but, as we show in §8.1, many interesting properties can be enforced with small overhead ($< 30\%$).

The nucleus' operation is transparent to the program and relies on intercepting (via instrumentation) all the calls made by the program that are relevant to the property enforced by the nucleus. Therefore, a nucleus is similar to the concept of execution monitors [7], except that instead of passively monitoring execution, it actively acts upon the program it is supervising and alters its behavior.

A nucleus is typically small, application-agnostic, and can be applied to multiple programs. Like most decisions in system design, the choice of what type of program properties are well suited for a nucleus is driven by experience and intuition. If the desired property can be expressed in terms of the system's execution trace, then it is a good candidate for a nucleus. Examples include security properties [14], resource accounting, etc. In §9 we discuss other types of properties amenable to iProve-style enforcement.

Instrumentation (§4) implements an abstraction function, which transforms any program's execution into an execution of the abstract program expected by the nucleus. Nucleus vendors prove the correctness of their nucleus using a third-party prover (e.g., Jahob [10]), as we show in §5.2. The same theorem prover is re-run on the nucleus' source code by the customer, to confirm the correctness of the nucleus.

The nucleus' correctness proof is usually decomposed into proofs of the nucleus' entry points, i.e., the methods invoked by the instrumentation code. The nucleus' join conditions can be expressed as $P = P_1 \wedge ... \wedge P_n$, where $P_i$ is a join condition. $P_i$ invokes entry point $E_i$, i.e., $P_i \Rightarrow E_i$[1]. The nucleus vendor proves that invoking $E_i$ provides property $Q_i$ to the target program ($E_i \Rightarrow Q_i$). Thus, if the instrumentation is correct, $P_i \Rightarrow Q_i$ because $P_i \Rightarrow E_i \wedge E_i \Rightarrow Q_i$. In order to prove that property $Q$ is enforced by the nucleus ($P \Rightarrow Q$), the nucleus vendor proves that $Q_1 \wedge ... \wedge Q_n \Rightarrow Q$, which is usually trivial.

iProve allows the use of easy-to-prove languages in the nucleus and easy-to-program languages in the body, thus affording ample flexibility in the choice of language. The nuclei we describe in §7 are written in Java or Prolog.

The programming language used for a nucleus impacts the trusted computing base (TCB). In iProve, the TCB includes the JVM, the layers beneath it, the iProve join verifier, and the nucleus verifier. For languages that cannot run inside a JVM, the library that mediates between the JVM and the nucleus' runtime must be included in the TCB.

It is possible for multiple nuclei to coexist and enforce composable properties, such as both encrypting communication and checking to not leak private information. iProve is conservative and considers two nuclei to conflict if they require the same instructions in $S$ to be instrumented. Generally, it can resolve this conflict statically, based on join conditions; if the conflict cannot be resolved, iProve does not continue. In general, nuclei can coexist if and only if the properties they enforce and their respective instrumentation code are orthogonal.

Assembling a proof for a sophisticated nucleus can take many days, even months. Proof-carrying code [12] showed that, even though generating a proof is a substantial burden, it makes sense for this burden to fall on the vendor. In our approach, provable code becomes more attractive, since nuclei are decoupled from the program bodies, so they can be reused across unlimited numbers of programs that desire the property enforced by the nucleus. This allows the cost of one expensive nucleus proof to be amortized across many uses of that nucleus in programs from many vendors.

[1] Whenever $E_i$ appears in a logic formula, we take it to mean the equivalent of statement "$E_i$ is executed."

## 4 The iProve Joiner

The purpose of the iProve joiner is to connect a nucleus to a target program, so that the program is controlled by the nucleus whenever necessary. Join conditions describe to the joiner and the verifier how the connection ought to be realized.

A join condition is a triplet $[\mathscr{B}, \mathrm{I}, \mathscr{A}]$. It states that, for every occurrence of instruction $I$ in the program, instruction sequence $\mathscr{B}$ must execute immediately *before* $I$ and instruction sequence $\mathscr{A}$ immediately *after* $I$. Join conditions are independent of program run-time state, and they apply to every occurrence of instruction $I$ in the program.

Join conditions abstract a program in a way that is suitable for proving the nucleus. The abstract program contains only the instructions relevant to the property the nucleus enforces. For example, for deadlock immunity, the program is treated as a sequence of lock and unlock operations. Such abstraction allows nuclei to be written in an application-agnostic fashion, making them easy to reuse.

Using join conditions, the nucleus developer specifies which program instructions are part of the abstraction. The $\mathscr{B}$ and $\mathscr{A}$ sequences transfer control to nucleus $N$ or relay information to it. For example, the join conditions for the deadlock immunity nucleus (§7.1) specify the `monitorenter` and `monitorexit` bytecode instructions to be of interest, as these correspond to the acquisition/release of locks in Java. In the case of a `monitorenter`, $\mathscr{B}$ invokes the nucleus function that decides whether to allow the lock operation to proceed or not, and $\mathscr{A}$ notifies the nucleus that the target lock has been acquired.

A second type of join condition is a triplet that is universally quantified over an inheritance hierarchy. Such a triplet specifies that the instruction of interest is a call to method $m$ of class $K$ *or* of any subclass of $K$. For example, the Java join conditions for the RSA-Crypt nucleus (§7.2) specify that calls to `java.io.OutputStream.write` are of interest *along with* all `write` calls in classes that inherit from `java.io.OutputStream`.

Finally, the third type of join condition is universally quantified over all execution paths, and it states control flow properties. One example is the requirement that a `monitorenter(x)` instruction must be followed on all execution paths by a `monitorexit(x)` instruction applied to the same object $x$. This type of join condition will typically have an empty $\mathscr{B}$ or empty $\mathscr{A}$.

The iProve joiner instruments the target program as required by the join conditions. Building a joiner is relatively straightforward, so we do not describe further design issues here. The only aspect of interest is the case where multiple nuclei are used at the same time. After applying join condition $P_1 = [\mathscr{B}_1, I, \mathscr{A}_1]$ and then $P_2 = [\mathscr{B}_2, I, \mathscr{A}_2]$, the resulting

program instrumentation will be $\mathscr{B}_1\mathscr{B}_2 I \mathscr{A}_2\mathscr{A}_1$. Thus, $P_2$ shifts $P_1$ away from $I$, causing $P_1$ to not hold any more, as it is no longer adjacent to $I$. The join verifier accounts for this shift as described in §5.1.

The iProve joiner need not be trusted. We note that it is easier to verify a posteriori the correctness of instrumentation than to guarantee a priori correct instrumentation.

The key element that enables iProve to scale to large systems is the decoupling of the nucleus from the body of the code. This enables the proof of a complex property on the small nucleus and a relatively simple instrumentation property on the body. This requires trusting the soundness of the join conditions (i.e., the nucleus is correctly invoked) and their completeness (i.e., all instructions of interest are covered). Incorrect join conditions void a nucleus-enforced property. Choosing appropriate join conditions is the responsibility of the nucleus vendor.

## 5 The iProve Verifier

The iProve verifier (Figure 2) consists of a join verifier (§5.1), which checks that the given join conditions have been correctly applied to system $S$, and a nucleus verifier (§5.2), which checks the correctness of the nucleus, independently of $S$. The iProve verifier is a distinctive feature of iProve compared to aspect oriented programming.

### 5.1 The iProve Join Verifier

Using a join verifier allows iProve to minimize the size of the trusted computing base (TCB); a smaller TCB means fewer assumptions, and therefore stronger guarantees. The amount of code in the join verifier is substantially less than in an instrumentation tool like AspectJ. Moreover, the join verifier uses algorithms that are simple and easy to manually check in an open-source implementation, thus making it easy to trust.

The join verifier's inputs are system $S$ and join conditions $[\mathscr{B}, I, \mathscr{A}]$. It checks that all relevant program paths satisfy the join conditions, thus soundly verifying the program/nucleus join. More specifically, for every method $m$ of the system $S$, if $m$ contains an instruction $I$ that is part of a join condition, then every execution path within $m$ that traverses $I$ must contain $\mathscr{B}$ before $I$ and $\mathscr{A}$ after it.

The join verifier algorithm uses a "bank account" data structure to detect violations of join conditions. A bank account has a savings and a debt component.

We can think of the control flow graph (CFG) of a method as a directed network of bank tellers, through which bank account ledgers are transferred from one teller to its successors. Each instruction in the method is viewed as a bank teller taking in bank accounts ledgers, performing

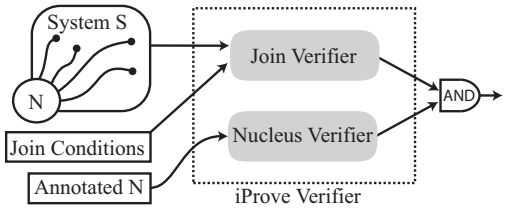



**Figure 2. Architecture of the iProve verifier.**

*transactions*, then forwarding them to another teller. The end goal is for there to be no debt in the system.

An edge $E = n_1 \rightarrow n_2$ in the CFG is labeled with the relation between its vertices as follows: *normal*, if the program execution flows normally from $n_1$ to $n_2$, or *exceptional*, if $n_2$ is executed within the handler of an exception thrown by $n_1$. A bank teller $b$ takes as input $in(b)$ a set of account ledgers from its predecessors, denoted by $pred(b)$. A bank teller's predecessors are grouped in two sets: $pred_n(b)$, containing predecessors $p$ such that the edge $E = p \rightarrow b$ is labeled as *normal*, and $pred_e(b)$, containing the other predecessors. The normal output of a bank teller, $out_n(b)$, is obtained by performing *transactions* on its input, i.e., $out_n(b) = transaction(in(b))$. However, $b$ can fail (i.e., if the instruction it models can throw an exception) and not perform the transaction—in this case, $b$ simply passes its input as exceptional output, $out_e(b) = in(b)$. Ledgers are transferred between bank tellers according to edge labels: for *normal* edges, $in(n_2) = out_n(n_1) = transaction(in(n_1))$, while for *exceptional* edges, $in(n_2) = out_e(n_1) = in(n_1)$.

Algorithm 1 describes the *transaction* operation. There are two types of instructions in a method: instructions of interest (targeted by join conditions) and normal instructions (the rest). We define $size(\mathscr{B})$ as the number of instructions present in the $\mathscr{B}$ component of a join condition. A bank teller processes its input ledgers one at a time (line 2). An instruction of interest (line 3) withdraws the top $size(\mathscr{B})$ instructions from the savings and compares them to those in $\mathscr{B}$ (line 4). If they match, the instructions in $\mathscr{A}$ are pushed onto the current debt (line 5). Normal instructions first try to cover a debt (lines 9-11). If there is no debt, they are pushed onto the savings (line 16). A new ledger is created with the new savings and debt (line 17). When a transaction cannot be performed (lines 7 and 13), an error is generated and the verification fails. Pushing an instruction on the debt requires the respective instruction to be the first one covered (i.e., popped off the debt stack).

The iProve join verifier employs an intra-procedural analysis shown in Algorithm 2. First, it adds to a worklist $W$ the bank tellers corresponding to all the instructions in the method currently analyzed (line 1). Initially, the input and output of every bank teller is an empty set of bank accounts (line 3), except for the one modeling the method's entry

**Algorithm 1**: The *transaction* algorithm

**Input**: program instruction $i$, set of ledgers $L$
**Data**: set of join conditions $J$
**Output**: new set of ledgers $L'$, or *error* if the transaction cannot be performed

1 $L' := \emptyset$
2 **foreach** $< savings, debt > \in L$ **do**
3  **if** $\exists < \mathscr{B}, i, \mathscr{A} > \in J$ **then**
4  **if** $top(savings) = \mathscr{B}$ **then**
5  $savings' := pop(\mathscr{B}, savings)$
 $debt' := push(\mathscr{A}, debt)$
6  **else**
7  *return error*
8  **else**
9  **if** $debt \neq \emptyset$ **then**
10  **if** $i = top(debt)$ **then**
11  $debt' := pop(debt)$
12  **else**
13  *return error*
14  $savings' := savings$
15  **else**
16  $savings' := push(i, savings)$
 $debt' := debt$
17  $L' := L' \cup \{< savings', debt' >\}$
18 **return** $L'$

**Algorithm 2**: The *ledger transfer* algorithm

**Input**: method $m$
**Data**: CFG of method $m$
**Output**: **true** if verification of the join conditions succeeds for method $m$, **false** otherwise

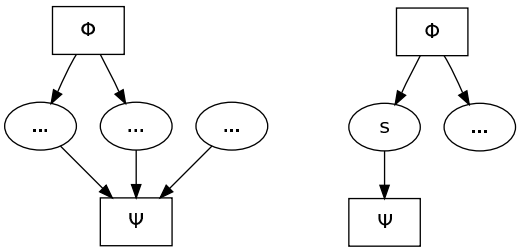
1 $W := instructions(m)$
2 **foreach** $i \in W$ **do**
3  $in(i) := out_n(i) := out_e(i) := \emptyset$
4 $in(start(m)) := \{< \emptyset, \emptyset >\}$
5 **while** $W \neq \emptyset$ **do**
6  $i := dequeue(W)$
 $in(i) := \bigcup_{p \in pred_n(i)} out_n(p) \cup \bigcup_{p \in pred_e(i)} out_e(p)$
7
8  $out_n^{old}(i) := out_n(i)$
9  $out_e^{old}(i) := out_e(i)$
10  $out_n(i) := transaction(i, in(i))$
11  $out_e(i) := in(i)$
12  **if** $out_n(i) \neq out_n^{old}(i) \vee out_e(i) \neq out_e^{old}(i)$ **then**
13  $enqueue(W, succ(i))$
14 **if** $\forall i \in exit(m) : debt(out_n(i)) = debt(out_e(i)) = \emptyset$ **then**
15  **return** true
16 **else**
17  **return** false

point, $start(m)$, whose input is a set with an empty bank account (line 4). Next, while there are elements in $W$ (line 5), the algorithm takes the first element in $W$ (line 6), and computes its input (line 7) and its output (lines 10 and 11). If the node's output changed since the last time it was visited (line 12), its successors are added to $W$ (line 13). Verification succeeds if the exit points $exit(m)$ of the method currently analyzed cover all debts (line 14) when $W$ is empty (i.e., all debts created by $\mathscr{B}$ have been covered).

When verifying multiple join conditions, all the debts created by all encountered instructions of interest must be covered for the analysis to succeed.

Join conditions that are specified over an inheritance hierarchy are verified using the same algorithm. These join conditions apply to methods calls only. To match a method call $K_j.m$ in a join condition to a method call $K_p.m$ in the program's code whose prototype matches, the join verifier analyzes the type relation between $K_j$ and $K_p$. If the type of $K_p$ is a subtype of $K_j$'s type, the match is successful. For example, in Java, if `Object.toString()` is of interest and the currently analyzed method has a call to `AnyClass.toString()`, the two instructions match, because `Object` is a superclass of all types.

The iProve verifier can check control flow properties. Consider two instructions of interest, $\phi$ and $\psi$; we want to express the fact that $\phi$ is always followed by $\psi$, but we do not want to constrain the number of program instructions spanned by this relation. Such relations are suitable for pairs of methods or pairs of operations, e.g., acquiring and releasing a lock. Figure 3a depicts this relation. In this property, $\psi$ acts as a debt whose maturity date is prolonged. However, for the verification to succeed, $\psi$ must eventually be covered. Debts with prolonged maturity dates must be covered in the reverse order of their creation.




(a) $\psi$ always follows $\phi$. (b) $\phi$ always precedes $\psi$.

**Figure 3. Verifiable control flow properties.**

A second type of control flow property is precedence of program instructions. For example, we may want to express that $\phi$ always occurs before $\psi$, as illustrated in Figure 3b.

While the property "$\psi$ always follows $\phi$" allows $\psi$ to appear without a preceding $\phi$, the property "$\phi$ always precedes $\psi$" forbids it. In this case, the join condition for $\psi$ marks $\phi$ as "anticipated" in $\mathscr{B}$. When $\psi$ is reached, the algorithm searches for $\phi$ in the savings part.

As an optimization, if it encounters a sequence of instructions not specified in any join condition and there is no debt, the iProve join verifier adds only the first instruction to the savings, thus reducing the corresponding ledger's size.

The iProve join verifier bounds the analysis to ensure the algorithm terminates. Loops in the CFG can cause the bank accounts to increase savings indefinitely. Every occurrence of an instruction of interest $I$ in the program body is called an "instance" of $I$ and noted as $I_p$. For the verification to succeed, every $I_p$ must be preceded by $\mathscr{B}$ and followed by $\mathscr{A}$, and this must be statically determinable. Hence, iProve does not accept join conditions that depend on the number of loop iteration. Thus, it is enough to analyze a single loop iteration to decide whether the join is done correctly or not.

### 5.2 The iProve Nucleus Verifier

iProve uses trusted third-party automated theorem provers to verify the correctness of nuclei. Our case studies (§7) rely on Jahob [10]. For some nuclei, verification may not even be necessary: for example, we used Prolog to write the privacy protector nucleus (§7.4), which can be treated as an executable formal specification.

The iProve verifier consists of the combination of the join verifier and the nucleus verifier—together they generate a modular proof that the systems instrumented with the nucleus will satisfy the stated property.

## 6 Implementation

We have built an iProve prototype that supports programs written in Java and nuclei written in Java and Prolog.

The iProve joiner uses AspectJ [2] to instrument programs. The join conditions are written as pointcuts and advices, as required by AspectJ. Pointcuts define the join points where the advice needs to be inserted. Advices specify what instructions to run before and after the pointcut.

The iProve join verifier uses instrumentation agents defined via the `java.lang.instrument` package to be invoked every time a class is loaded. iProve employs load-time verification because, due to the advent of load-time instrumentation, verifying the persistent representation of a class does not guarantee the analysis' results will still hold at run time. This approach ensures that the verification of join conditions is done on the code that is actually run.

iProve verifies both the program and all linked libraries, i.e., it all the classes the program can interact with, including the JDK classes. It is crucial for iProve to do so, because nuclei-provided properties can be voided by improperly instrumented libraries or already loaded classes.

The iProve join verifier uses a cache of previously verified classes to avoid redundant work. Prior to analyzing a class, the join verifier checks the cache, based on a hash of the class' bytecode. If found, the class is no longer verified. Experimental evidence shows that using a cache considerably speeds up system verification.

iProve relies on the JVM bytecode verifier to enforce correct types. iProve checks for instrumentation correctness, which is orthogonal to what the JVM's bytecode verifier checks: language construct violations, references to fields and methods, correct use of types, etc.

## 7 Four Case Studies

We wrote several iProve nuclei. In this section, we present four of them: deadlock immunity (§7.1), secure communication (§7.2), bounding of resource consumption (§7.3), and privacy protection (§7.4). These four case studies illustrate a wide range of provable properties.

### 7.1 Dimmunix: Deadlock Immunity

Deadlock immunity [8] is a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of that deadlock. The Dimmunix nucleus relies on program instrumentation to intercept all lock/unlock operations, watches for deadlocks at runtime and, when they occur, adds a corresponding fingerprint to a persistent history file. If Dimmunix observes that the instrumented program is headed toward a deadlock whose fingerprint is in the history, it schedules threads so as to avoid the deadlock. Augmenting programs with this property helps consumers cope with hard-to-fix deadlocks. iProve can be used by software vendors to prove that their (potentially complex) software has deadlock immunity.

We wrote the nucleus in $\sim$600 lines of Java, and used Jahob to prove its correctness. Writing this relatively complex nucleus taught us that, given today's provers, knowledge of the intended proof methodology is crucial to writing the code appropriately. This further strengthens our belief that isolating desired properties into nuclei that are independent of the main body of code is the only tractable way to prove properties of real software today.

The join conditions require the nucleus' method *beforeLock(x)* to be called before each `monitorenter(x)`. After acquiring the lock, Dimmunix must be notified via *afterLock(x)*. Before `monitorexit(x)`, *beforeUnlock(x)* needs to be called.

Despite the extensive effort involved in proving this nucleus (8 developer-weeks), we believe it was worthwhile,

since it can be reused across an unlimited number of Java programs. As we show in §8, we applied the Dimmunix nucleus to real systems, including JBoss, a system that cumulatively has >3 MLOC. Access to these systems' source code was not required. Runtime performance overhead introduced by the Dimmunix nucleus is less than 30%.

### 7.2 RSACrypt: Secure Communication

The second nucleus is targeted at securing the communication between systems. The RSACrypt nucleus is written in Java and encrypts/decrypts using RSA all data sent/received via Java I/O mechanisms.

The secure communication property is merely a mathematical statement that all messages sent are suitably encrypted according to the RSA algorithm (i.e., if message $M$ is written, then $M^e \bmod n$ is actually sent out, where $(e,n)$ is the recipient's public key) and all messages received are suitably decrypted. Like for Dimmunix, we used Jahob to prove the nucleus' correctness.

The join conditions state that each call to `write(x)` in `OutputStream` and any classes that extend `OutputStream` must be preceded by a call to the nucleus' method *encrypt(x)*, and similarly for decryption on the inbound `InputStream.read`. If the target program uses custom I/O mechanisms, the join conditions must also cover these mechanisms.

### 7.3 CPUBound: Resource Utilization

This nucleus guarantees that a target Java system will not utilize CPU time more than a given percentage during any 1-minute interval. The nucleus throttles the application's threads when CPU consumption approaches the threshold: it checks CPU utilization periodically and decides whether it is safe to let the application continue running with all its threads. If doing so presents a risk that the threshold will be exceeded for the remainder of the current 1-minute sliding window, the nucleus temporarily suspends the most active threads. Threads are resumed when there is no risk of overstepping the CPU time limit. To prevent thread starvation, the nucleus performs a round-robin scheduling of threads every minute: suspended threads are resumed and other threads are suspended instead.

It is imperative that the nucleus thread be started before all the other threads. We therefore use a special join condition which specifies that the first instruction in the program's `main` function is a call to the nucleus. We employed Jahob to prove the correctness of the CPUBound.

### 7.4 PrivacyGuard: No Data Leaks

The final nucleus is a "firewall" meant to prevent programs from accidentally leaking private information.

We used iProve in hybrid mode: we targeted Java programs, but wrote the nucleus in Prolog—a programming language that is a direct representation of formal logic. Prolog programs contain Horn clauses that represent a subset of first-order predicate logic; Prolog clauses are either "facts" about the world, or "predicates."

What is interesting about writing a nucleus in Prolog is that, being itself written in first-order logic, the program can be treated as an executable formal specification. This illustrates the power of supporting nuclei written in languages different from the main body: they can be written in a way that is well suited for proving, while the bodies can be written in languages suited for the task of interest.

The join conditions target the same write instructions as for the RSACrypt nucleus, and connect the Java program to the Prolog environment via the JPL extension [6]. The instrumentation invokes the nucleus immediately prior to sending data and translates the Prolog query results into a buffer that is passed down to the JDK for sending.

## 8 Evaluation

In this section, we systematically evaluate iProve, emphasizing consumer-visible aspects. In §8.1 we show that iProve can be used to verify interesting properties in large Java systems. We show it is practical to have the join verifier in the critical path of running all software on a system. In §8.2 we show it is practical to construct and verify nuclei, and that the added effort is worth the promise of software guarantees.

Table 1 shows information on the systems we used in our evaluation. JBoss is an enterprise application server, Limewire is a peer-to-peer file sharing application, and Apache ActiveMQ is a message broker. We show the number of classes, methods, and LOC in the standard distribution package. This includes only classes that are directly checked by iProve; due to packaging artifacts, the same class may appear (and be verified) more than once, e.g., when it is part of statically linked libraries.

| System | # Classes | # Methods | LOC |
|---|---|---|---|
| JBoss-4.0.2 | 34,081 | 269,598 | $3.17 \times 10^6$ |
| Limewire-4.18 | 19,176 | 138,990 | $1.51 \times 10^6$ |
| Apache ActiveMQ-4.0 | 1,865 | 15,568 | $1.33 \times 10^5$ |

**Table 1. Real-world software systems used to evaluate iProve.**

### 8.1 Performance

Table 2 summarizes how long the instrumentation verification takes. After the program is executed for the first time, the user will not notice any substantial delays, since only modified classes will be re-verified. The *first execution* time includes verifying auxiliary classes (e.g., JDK or AspectJ classes). The third column shows the latency a user would normally observe; the majority of the time is spent computing the hashes of loaded classes. The results validate our design choice to use a cache. The time needed to verify the nuclei (§8.2) must be added to the reported verification time, but only marginally increases the reported times.

| System | Time to verify at ... | |
|---|---|---|
| | First execution | Subsequent executions |
| JBoss-4.0.2 | 23 min 18 sec | 14 sec |
| Limewire-4.18 | 3 min 18 sec | 71 sec |
| Apache ActiveMQ-4.0 | 2 min 59 sec | 19 sec |

**Table 2. Time to perform join verification.**

To assess the runtime performance overhead introduced by nuclei, we measured JBoss with the RUBiS e-commerce benchmark [13]. There are no standard benchmarks for Limewire and ActiveMQ, so we ran benchmarks included with these systems: upload throughput and client-side throughput tests for Limewire, and JMS queue throughput tests for ActiveMQ. We exercised each system with every nucleus in isolation and also with an ensemble of nuclei comprising Dimmunix, RSACrypt, and CPUBound. Table 3 summarizes the results. Overhead is mostly below 30%, because the iProve join verifier always analyzes the minimum set of instructions necessary to validate the property. The one notable exception (JBoss with the 3-nuclei ensemble) is due to the CPU consumption limitation imposed by the CPUBound nucleus: it throttles the CPU-intensive RSACrypt nucleus.

Having seen the end-to-end results in real systems, we now focus on characterizing in detail the performance of the iProve verifier. In particular, we evaluate the join verifier's performance as a function of the number of join conditions. The more join conditions are in the specification, the more work the iProve verifier has to do. Figure 4 shows the results. We wrote 1 to 50 join conditions targeting different instructions of interest. Each join condition specifies a method that needs to be invoked before the instruction of interest and another one after. For each number of join conditions *n*, we wrote a class that contains 10 methods. Each method contains a permutation of the *n* instructions of interest, with code that contains loops in between join conditions. The results show that the iProve join verifier scales well with the number of join conditions it needs to verify.

| Nucleus | JBoss | Limewire | ActiveMQ |
|---|---|---|---|
| Dimmunix | 13.0% | 27.2% | 3.1% |
| RSACrypt | 15.1% | 13.4% | 0.7% |
| PrivacyGuard | 10.0% | 7.3% | 1 .0% |
| CPUBound | 0.0% | 3.9% | 0.1% |
| CPUBound & Dimmunix & RSACrypt | 118.4% | 32.1% | 3.9% |

**Table 3. Runtime overhead of single nuclei and an ensemble of 3 nuclei.**

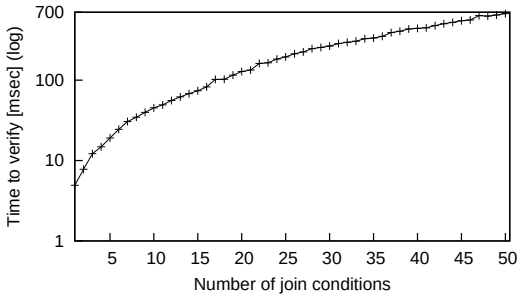



**Figure 4. Verification time as a function of the number of join conditions.**

### 8.2 Effort to Write and Prove Nuclei

Table 4 quantifies the effort and complexity of developing and proving the nuclei reported in this paper. All the nuclei were written in Java, except PrivacyGuard, which was written in Prolog. We consider the results to be representative of what one could expect in practice. As mentioned before, the investment in developing formally proven nuclei can easily be amortized across multiple programs.

| Metric | *Dimmunix* | *RSACrypt* | *CPUBound* | *PrivacyGuard* |
|---|---|---|---|---|
| Time to develop the nucleus | 1 week | 6 hrs | 2 days | 4 hrs |
| Time to develop the proof specification | 8 weeks | 3 days | 2 days | N/A |
| Nucleus size (LOC) | 607 | 73 | 180 | 8 |
| # of annotations | 253 | 24 | 47 | N/A |
| # of assumptions | 38 | 0 | 17 | N/A |
| Proof complexity (# of proof obligations) | 2668 | 179 | 264 | N/A |
| Time (for Jahob) to prove nucleus | 121 sec | 3.3 sec | 7.7 sec | N/A |

**Table 4. Costs of writing and proving nuclei.**

## 9 Discussion and Limitations

iProve provides a bridge between static analysis and formal proofs. The properties enforced by the four nuclei presented here cannot be checked with pure static analysis, so they require a runtime component. However, only providing the runtime component cannot guarantee that it will always be correctly invoked, or that it is free of bugs. iProve pieces together the static and runtime parts in a way that can be proven to achieve the desired property.

In theory, it would be ideal for nucleus vendors to also provide a proof that joining system *S* with nucleus *N* does not alter *S*'s functionality. In many cases, the orthogonal nature of the iProve nucleus makes it easy to check (e.g., as in the case of encrypting/decrypting incoming/outgoing network traffic). In less obvious cases, the most practical approach is for software vendors to use their existing tests to validate their program's functionality, and then use iProve to prove to consumers that the program they are about to install or run has the desired properties.

The current version of iProve assumes the JDK, the JVM, and the OS are correct. In other words, the trusted computing base includes the layers below the program and iProve. Without iProve, the program itself would have to be part of the TCB; with iProve, the consumer is no longer required to trust the program.

Not all properties are suitable for verification with the iProve approach, and this is a limitation that results directly from the nucleus/body partition we propose. For example, a nucleus cannot enforce liveness properties.

Properties targeting thread scheduling, like signal/wait pairing, can be enforced with iProve. A nucleus will store information about incorrect behavior and modify the thread schedule to avoid it. If a property can be expressed as a pure function, then it can also be an iProve nucleus. Prevention of SQL injection attacks, for instance, can be cast as a pure function that, given a string representing the SQL query to be executed, removes any parts of *WHERE* clauses that could turn into tautologies (e.g., *WHERE ... OR 't'='t'*).

A nucleus can enforce properties related to the program/environment interaction. An example is a nucleus that enforces a given policy for accessing resources, such as files or network ports. Although implementable using other approaches as well, using iProve makes the work reusable across several different programs and OSes.

For system developers, using iProve requires that they architect their software slightly differently. There is strong synergy between the iProve approach and both aspect-oriented programming and agile development, both of which advocate decoupled designs. iProve proposes a system architecture in which non-functional properties are externalized as formally proven nuclei that exercise control over program features. This decoupled design allows each component of the program to focus on its primary goal.

Finally, whereas most of the static analyzers, theorem provers, or model checkers require access to source code, iProve only needs access to the nucleus' source code. This permits software vendors to provide consumer-verifiable guarantees without disclosing their proprietary code.

## 10 Related Work

iProve aims to enable software with guarantees. In so doing, we build upon a rich history of prior work, with **proof-carrying code** (PCC) [12] having inspired us the most. Whereas PCC could prove mostly low-level properties, like type and memory safety, iProve proves higher-level properties, like deadlock immunity. In iProve, unlike PCC, a change in the program (e.g., a software update) does not require redoing the nucleus proof—the nucleus can be reused across unlimited versions of the same program. iProve differs from PCC in two main aspects: First, in PCC, one needs to provide a proof for the entire system, whereas in iProve only a proof of the nucleus and the verification of simple join conditions are needed. Second, in iProve, the nucleus is automatically verified using a theorem prover; in PCC, the proof is usually written manually, which is significantly harder than providing hints to an automated theorem prover.

The idea of concentrating code that enforces properties in a single component was pioneered by **reference monitors** [1] and implemented in Data Secure Unix (DSU) [17]. There, the efforts were focused on formally specifying and verifying that a security kernel has the mechanisms to enforce any security policy. iProve can be thought of as using the join conditions to abstract a program into a security kernel, and the join verifier checks that the kernel has the mechanisms to enforce a property. DSU's policy manager could be an iProve nucleus. While the security kernel can be used only with DSU, iProve nuclei work for all Java programs.

iProve uses **aspect oriented programming** [9] to ease the writing join conditions. While AOP allows program semantics to be altered, an iProve nucleus is not allowed to do that. The iProve join verifier is complementary to AOP: it checks that the instrumentation performed by an AOP compiler is correct.

The idea of proving properties of an aspect in isolation appeared in SuperJ [15], which proved that an aspect provides a certain property *F* to any system. SuperJ needs a generic program that captures all possible behaviors relevant to the aspect: if *F* holds for the generic program instrumented with the aspect, then *F* holds for any system instrumented with the same aspect. One can use model checking techniques to prove that *F* holds for the generic program. The drawback of this approach is that the generic program must be written manually.

JavaMOP [3] defines **runtime monitors** to supervise program properties at runtime. When a property is violated, the monitor can perform error recovery actions, but these actions are system-dependent. In contrast, an iProve nucleus is system-independent and prevents the program from violating the desired property.

Runtime verification systems, like Java Path Explorer [5], can check high-level system properties in large applications, in the same way iProve can; these properties are often expressed as temporal logic formulae. These systems perform only passive runtime monitoring, to check if some property holds; they do not enforce the property.

The systems presented in [11] and [16] can be implemented as iProve nuclei. The communication between a distributed system's nodes can be augmented with timing-related properties by using [11]. In [16], on every method call, an integrity kernel checks if the caller object has the right to access the callee object, to prevent usage of untrusted data by critical parts of the system.

The iProve join verifier is similar to ESP [4]; however, there are some important differences between the two. First, ESP checks program-wide properties, while the iProve join verifier only needs to check local intra-procedural properties. Second, the iProve join verifier also handles exception control flows, whereas ESP does not.

## 11 Conclusion

iProve is a technique for formally verifying complex program properties in systems with over a million lines of code. Desired properties are proven as a combination of two proofs: one of a complex property applied to a nucleus and one of a simple property applied to the program body. We demonstrated iProve's feasibility through four case studies, in which nuclei enforce deadlock immunity, secure communication, resource usage bounds, and dataflow control in three real systems: JBoss, ActiveMQ, and Limewire. Our experiments show that iProve can verify large systems in minutes. iProve requires no access to program source code and allows nuclei to both be reused for unlimited numbers of systems and to be written in verification-friendly languages. Thus, iProve enables developers to produce and distribute software with customer-verifiable guarantees.

## Acknowledgments

We are indebted to Pranav Garg for his work on proving the Dimmunix nucleus. We thank our shepherd Keith Marzullo, the anonymous reviewers, and our EPFL colleagues for their help in refining our paper. We are especially indebted to Viktor Kuncak for his help with Jahob.

## References

[1] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, ESD/AFSC, 1972.

[2] AspectJ. http://www.eclipse.org/aspectj.

[3] F. Chen and G. Rosu. Java-MOP: A monitoring oriented programming environment for Java. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[4] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Conf. on Programming Language Design and Implementation*, 2002.

[5] K. Havelund. An overview of the runtime verification tool Java PathExplorer. In *Formal Methods in System Design*, 2004.

[6] Java-Prolog. http://sourceforge.net/projects/jpl, 2009.

[7] C. Jeffery, W. Zhou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. In *Workshop on Program Analysis for Software Tools and Engineering*, 1998.

[8] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Symp. on Operating Systems Design and Implementation*, 2008.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conf. on Object-Oriented Programming*, 1997.

[10] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS, MIT, 2007.

[11] P. Martins, P. J. P. de Sousa, A. Casimiro, and P. Verissimo. Dependable adaptive real-time applications in wormhole-based systems. In *Intl. Conf. on Dependable Systems and Networks*, 2004.

[12] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Symp. on Operating Systems Design and Implementation*, 1996.

[13] RUBiS. http://rubis.objectweb.org, 2007.

[14] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[15] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 2003.

[16] E. Totel, J. P. P. Blanquart, Y. Deswarte, and D. Powell. Supporting multiple levels of criticality. In *Intl. Symp. on Fault-Tolerant Computing*, 1998.

[17] B. J. Walker, R. A. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix Security Kernel. *Communications of the ACM*, 23(2), 1980.

[18] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *Conf. on Programming Language Design and Implementation*, 2008.