# Proof-Carrying Apps: Contract-Based Deployment-Time Verification*

Sönke Holthusen, Michael Nieke, Thomas Thüm, and Ina Schaefer

Institute of Software Engineering and Automotive Informatics
TU Braunschweig, Germany
{s.holthusen, m.nieke, t.thuem, i.schaefer}@tu-bs.de

**Abstract** For extensible software platforms in safety-critical domains, it is important that deployed plug-ins work as specified. This is especially true with the prospect of allowing third parties to add plug-ins. We propose a contract-based approach for deployment-time verification. Every plug-in guarantees its functional behavior under a specific set of assumptions towards its environment. With *proof-carrying apps*, we generalize proof-carrying code from proofs to artifacts that facilitate deployment-time verification, where the expected behavior is specified by the means of design-by-contract. With proof artifacts, the conformance of apps to environment assumptions is checked during deployment, even on resource-constrained devices. This procedure prevents unsafe operation by unintended programming mistakes as well as intended malicious behavior. We discuss which criteria a formal verification technique has to fulfill to be applicable to proof-carrying apps and evaluate the verification tools KeY and Soot for proof-carrying apps.

**Keywords:** deployment-time verification, design-by-contract, software evolution

## 1 Introduction

A recent development in software development is the consumers' need for customization which can be achieved by several approaches [28]. For instance, frameworks with plug-ins allow for customization and reduction of maintenance effort by code reuse. Functionality does not have to be implemented several times such that developers are able to reduce their code base. Prominent examples are mobile app platforms for smartphones and tablets running on Android and iOS, where users can extend the functionality of the operating system by installing custom apps, such as messengers or games. However, end-users installing apps and plug-ins also poses risks as additional, potentially malicious, code is added. On smartphones, misbehaving third-party apps can be mainly considered a security risk, e. g., by stealing sensitive data. If extensible platforms are also used in other domains, such as the automotive domain, a misbehaving app might lead to physical damage and even injuries to persons. This is especially relevant when the car producers are eventually opening their platforms for

* This work was partially supported by the DFG (German Research Foundation) under the Researcher Unit FOR1800: Controlling Concurrent Change (CCC).

third party plug-ins with access to safety-critical functions like steering or controlling the speed of the car, e. g., advanced driver assistance systems like adaptive cruise control. To support execution of apps within a car, it is mission-critical that the program logic utilizes provided safety-critical interfaces as they are intended to be used, e. g., by abiding by the contracts or by following a certain protocol.

Defensive programming [20] can catch unexpected behavior at runtime, but it cannot prevent it. For example, a car could enter a fail-safe mode, but such situations should be avoided whenever possible. Another way to ensure certain program properties is the use of formal methods, which make it possible to prove that a specific behavior will or will *not* happen. While some program properties can be verified automatically with verification techniques such as theorem proving [27] and software model checking [5,15], interaction is necessary for other properties and verification techniques. However, interaction is impractical in deployment-time verification.

To guarantee global safety properties for assembly language, Necula introduced the concept of proof-carrying code [22]. Under the assumption that the proof generation is harder than checking a proof, the verification of properties is divided into two distinct phases: In the first phase, a proof is created, possibly with user interaction. In the second phase, the proof is merely checked at the code consumer in order to speed up the verification process. Proof-carrying code can handle global safety properties, while extensible platforms call for the ability to support interface-specific specifications. Hence, proof-carrying code has to be generalized to ensure program properties in the context of extensible platforms and to allow local specifications. In this paper, our contributions are the following:

- With proof-carrying apps, we propose a generalization of proof-carrying code to deployment-time verification of apps. In particular, we generalize proof-carrying code from deductive verification to other static verification techniques and generalize proofs to any artifact that simplifies verification, whereas we specify intended behavior by means of contracts (see Section 2).
- We discuss criteria for verification techniques that enable their use for deployment-time verification with contracts. We discuss hard criteria, such as the need to run completely automatic, and soft criteria, such as a considerable speed-up due to proofs or other artifacts (see Section 3).
- We compare deductive verification, data-flow analyses, and software model checking for their advantages and limitations for proof-carrying apps. Whereas contracts are typically checked with deductive verification, we propose how to check contracts with data-flow analyses and software model checking (see Section 4).
- We investigate the applicability of deductive verification with KeY and data-flow analyses with Soot for proof-carrying apps (see Section 5).

## 2 Proof-Carrying Apps

Proof-carrying code was introduced to automatically ensure that a program written in assembly language complies to a specific set of safety policies [22]. This is achieved by delivering a proof with the executable and checking it before the program is executed. The principle is automated by a certifying compiler for C and Java, generating machine code and the required certificate [11, 23]. The MOBIUS project applied proof-carrying code to Java and information flow properties [6]. *Proof-carrying apps* are influenced by those approaches.

## 2.1 Proof-Artifacts for Apps

With proof-carrying apps, we extend proof-carrying code to be used in extensible software platforms, such as black-box frameworks with plug-ins. As an abstraction from the concrete extensible system, we use plug-ins as units of extension, but the general concepts of this paper may be adapted to other approaches as well. A framework provides extension points that can be extended by plug-ins, which itself can provide further extension points to other plug-ins [3].

One way to specify what is necessary for a plug-in to work in an environment of other plug-ins is to capture its assumptions and guarantees towards this environment. We are interested in verifying a plug-in against the (observable) behavior of the other plug-ins and need an appropriate abstraction of the behavior which can be understood as guarantees towards the environment. Additionally, we want to be able to restrict the usage of an API, which is an assumption towards the environment that no plug-in uses the API otherwise. To handle assumptions and guarantees, we use contracts where assumptions are preconditions and guarantees are postconditions [20]. Contracts also allow us to express explicitly who is responsible in the case of an error. The calling plug-in is responsible to ensure the precondition of the called plug-in and therefore for possible errors due to violated preconditions. In that case, the behavior of the called plug-in can not be guaranteed. Developers may use all services provided by available plug-ins and the runtime environment. Furthermore, the contracts abstract from the implementation of a plug-in and form the basis for proof-carrying apps.

Figure 1 shows the basic concept of proof-carrying apps. The platform developers provide contracts for their API. In the first phase, developers write their plug-in, and proof artifacts are created with proof generators and possibly necessary interaction with dedicated experts. The proof artifact supports the developers' claim that their plug-in complies with the contract. Instead of the proof (i. e., certificate) in proof-carrying code, we allow every kind of *(proof) artifact* supporting the verification of a contract. Depending on the used verification technique, manual interaction with the verification tool might be required, e. g., when providing a loop invariant for the proof or applying certain tactics. In the second phase, after the proof artifact is generated, it can automatically be checked by a proof checker using the plug-in and the proof artifact. All the information provided manually in the first phase, such as loop invariants, will be reused, resulting in a check without interaction being necessary. If the specification holds using the proof artifact, the plug-in can be executed safely with respect to the contract. In the case, the contract does not hold, a possible solution is that the plug-in is not allowed to be executed, and the user receives a warning.

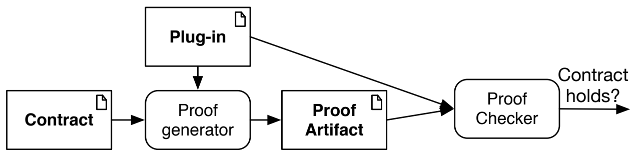

**Figure 1.** The basic proof-carrying app concept.

## 2.2 Application Scenarios

To further motivate our concept of proof-carrying apps, we introduce two application scenarios. The first uses on-device validation, while the second involves a trusted third party for validation purposes.

*Scenario 1: On-Device Validation*

1. The development team writes the plug-in and creates a proof artifact to support the claim that all relevant specifications are met.
2. The plug-in and the corresponding proof artifact are deployed onto the device.
3. The device uses a trusted proof checker to check whether the provided plug-in complies with the specification using the proof artifact.

This scenario has the advantage of only requiring a very small *trusted computing base* [26]. The only component to be trusted is the proof checker on the device but as it is running on the device the checker has to be small enough and the verification on the device should be much more efficient than those in the first phase. As an extension to the first scenario, we add a trusted third party to our trusted computing base in the second scenario.

*Scenario 2: Validation at a Trusted Third Party*

1. The development team writes the plug-in and creates a proof artifact to support the claim that all relevant specifications are met.
2. The plug-in and the corresponding proof artifact are submitted to the trusted third party.
3. The *trusted third party* (e. g., the framework operator) checks the proof using a trusted proof checker.
4. If the check is successful, the program is marked as safe, signed cryptographically by the trusted third party, and deployed to the device.
5. On the device, only the signature has to be checked, and the plug-in is executed, if the signature was validated.

The app stores of Android or iOS for smartphones are an example where this scenario could be realized. App developers could submit their plug-in and proof artifacts to the app store where their app is checked. If the proof holds, the app is marked as safe, and it can be downloaded from the store onto the device.

## 3 Criteria for Deployment-Time Verification with Proof-Carrying Apps

In this section, we discuss the criteria that a verification technique has to fulfill to work with the validation scenarios introduced in Section 2. These criteria help to find verification techniques suitable for proof-carrying apps and give a means to compare different techniques. To be usable without user interaction, a proof-carrying app has to be checked *fully automatically* on deployment. Considering that the resources are very limited on device, the time for checking whether the contracts hold using the proof artifact should be *reasonably fast*. We do not want to trust the developers of an app and, hence, cannot trust artifacts deployed by them. In the following, we distinguish between hard criteria being mandatory and soft criteria being desirable.

*Automatic Verification* The most important, hard criterion of a verification technique is its ability to check a proof fully automatically. It is not reasonable to assume that users (e. g., drivers of a car) have any knowledge of any verification techniques. While the generation process of the proof artifact may include interactions with developers, the check has to be done without requiring any input. If input is necessary for a successful verification, e. g., in the form of invariants, the technique has to provide a way to store this information for a later automatic reuse. One example are proof scripts which record every step and every decision necessary to reproduce the proof. Every formal verification technique to be considered has to be able to check proof artifacts fully automatically. Furthermore, the proof checker has to be part of the trusted computing base. Thus, has to be sound, and its footprint has to be comparably small to run on embedded devices.

*Reduced Verification Effort for Deployment* In the *on-device validation* scenario, the proof checking takes place on the target device, which usually has limited resources including processing power. To reduce the verification effort on the device, we are looking for verification techniques which support a considerable speed-up for proof artifact checking, opposed to proof artifact creation. One possibility to achieve this is by generating an artifact in a first validation run to speed up a second run (in the proof artifact checker). Existing techniques exhibiting this behavior are verification techniques supporting evolution or incremental proofs. Usually the entire program is verified every time. This can be a time consuming task, even for very small changes to the source code. Verification techniques supporting evolution can reduce the necessary resources. The source code has to be verified completely once and the proof itself or an artifact helping with checking the proof is saved. The following verification run can reuse this artifact to reduce the source code which has to be verified. When the source code is modified, the differences between the two versions are calculated. It is then determined whether the changes have an impact on the validity of the proof. If it is necessary to change or redo the proof, it can be generated using the old proof artifact and the source code delta. For proof carrying apps, we assume that for this kind of verification approach, the verification effort of two *identical* programs should be minimal. The result of the first verification is saved as proof artifact and deployed with the plug-in to be used for the proof check at deployment time. Because the plug-in does not change between the two verifications, the second pass should be considerably faster. The reduction of the verification effort is a soft criterion.

*Trust in Proof Artifacts* For our concept of proof-carrying apps, the developers of apps have to generate and provide the proof artifacts with their app. The proof artifacts are used in the check whether the contracts hold or not. One way to prevent a negative result of this check, could be to modify the proof artifact. Therefore, we do not want to trust the developers and the proof generator and we need verification techniques which do not need to trust generated artifacts for the check. The absence of trust towards the generated proof artifacts is a hard criterion.

## 4 Formal Verification Techniques

In the previous section, we introduced a set of criteria a verification technique should fulfill for deployment-time verification with proof-carrying apps. To determine whether a technique is suitable for proof-carrying apps, we give a short comparison of three verification techniques and discuss how contracts can be verified by each of them.

### 4.1 Data-Flow Analysis

Data-flow analysis is a method for static analysis. At the time of compilation, it calculates an approximation of how a program will behave at runtime [24]. Following the control flow of a program, data-flow analysis can give information about what values a variable at a specific statement in a program may have. The approximation allows data-flow analysis to reach a fixpoint without needing invariants for otherwise undecidable loops or recursion. Data-flow analysis can run automatically, but as a drawback it can return false positives (i. e., correct programs marked as incorrect) and false negatives (i. e., erroneous programs marked as correct). To guarantee that contracts hold, we cannot allow false negatives to occur. The analysis has to be strict enough to catch all possible false negatives. In contrast, the presence of false positives has no influence on the safety of an app, but the proof checker would mark a safe app as unsafe, and the app would not be allowed to run. Hence, we would need to reduce the number of false positives to zero, e. g., by refactoring the source code or by providing loop invariants or assertions. The Clang Static Analyzer[1] uses data-flow analyses to find errors in C/C++ programs while Soot provides data-flow analyses for Java [16].

*Contracts* Data-flow analysis tools usually do not work with contracts but with data-flow problems like *reaching definitions* analysis. This analysis gives a list of variable definitions which may influence the value of variables at a specific point of the program. However, we can encode pre- and postconditions into data-flow problems, by translating them into runtime assertions. When a method with a contract is called, the calling method is responsible that the precondition is not violated. Hence, before entering the method, we have to add an explicit check for a violation

[1] http://clang-analyzer.llvm.org/

using an if-statement encoding the precondition. If the precondition is violated, a *PreconditionViolationException* is thrown. The called method is responsible that the postcondition is not violated if the precondition holds. The postcondition is checked at the end of contracted method and throws a *PostconditionViolationException* if the check fails. The data-flow analysis has to verify that, for the program under verification, the statement throwing the exception cannot be reached.

### 4.2 Software Model Checking

A software model checker uses a combination of different verification approaches to prove properties of programs [15]. This includes abstract interpretation, predicate abstraction, and state-space exploration. A software model checker returns either that the property holds for the input model or it gives a counterexample which can help to find the error. With the right encoding of the property, a model checker can run automatically. The biggest drawback of model checking is the limited scalability, as it tends to run out of main memory. Modern model checkers have numerous parameters to improve scalability, such that good settings need to be found for verification. However, those settings could also be passed to the device for deployment-time verificaiton. Java PathFinder[2] is a software model checker for Java [14] and CPAchecker supports model checking of programs written in C [8].

*Contracts* The encoding for a software model checker can be similar to that for the data-flow analysis. If the precondition is not satisfied before the call of a contracted method, an exception is thrown. Before the contracted method returns, the postcondition is checked, and an exception is thrown if it is not satisfied. To ensure that pre- and postconditions are satisfied, the model checker has to check that none of the exceptions are thrown.

### 4.3 Deductive Program Verification

Theorem provers use deductive reasoning to prove logical properties [27]. Techniques like symbolic execution (e. g., KeY [1]) and verification-condition generation (e. g., Dafny [19]) are also utilized. The input is processed with the help of inference rules. Due to situations in which more than one rule or different instantiations are possible, user input might be necessary. To avoid decidability problems with loops or recursion, it also might be necessary to provide invariants. Both properties make theorem proving often not a fully automatic technique. Nevertheless, once a proof is completed, it can be stored and automatically checked at deployment-time. However, it is not clear whether such proof replay is significantly faster than proof finding. Deductive program verification tools supporting proof replay are, e. g., KeY [1] or Coq [7].

*Contracts* A theorem prover translates the program and contracts to a calculus. Pre- and postconditions can be transfered directly into logical expressions of that calculus. Such expressions can reference variables used in the program and can be added as an annotation to the contracted methods. Invariants can be added to loops or classes.

[2] http://babelfish.arc.nasa.gov/trac/jpf

## 5 Comparison of Verification Techniques

After introducing verification techniques which support fully automatic proof checking, we investigate which tools satisfy the aforementioned criteria (cf. Section 3). For our comparison and the case study, we decided to focus on the Java programming language. It is a widespread language (e. g., Android) and the support for verification tools is in general quite good. The *KeY tool* [1] is a theorem prover and has the ability to load and replay proofs once they are found. The second tool, *Soot* [16], uses data-flow analysis and was extended to support incremental changes [4]. Both tools support a way to reduce the verification effort, KeY by using proof scripts and Soot by utilizing incremental analysis, which is the reason we chose them. *Java PathFinder* [14] is the most prominent example for a software model checker supporting Java. Unfortunately, we were unable to find work for supporting evolution or incremental analysis in Java PathFinder or other model checkers for Java.

### 5.1 Case Study

We first present the basic program to be verified before we show how it was adapted for KeY and Soot. We introduce a Java framework representing a car control with the interface shown in Listing 1.1. The interface `CarControl` allows setting a target speed for the car by using method `setTargetSpeed`. It also allows reading the current speed by using method `getCurrentSpeed`. The interface is annotated with contracts in JML [18] which is used for specifications in KeY. JML annotations are distinguished from normal comments by adding an @ as shown in Listing 1.1. The annotation of method `setTargetSpeed` in interface `CarControl` consists of one precondition. In this case, the value of the argument `speed` is assumed to be larger or equal to 0 and smaller or equal to 300. The interface developer has to provide the contracts for the interface.

```
public interface CarControl {
/*@
@ public normal_behavior
@ requires speed >= 0 && speed <= 300;
@ assignable \nothing;
@*/
  public /*@ helper @*/ void setTargetSpeed(int speed);
/*@
@ public normal_behavior
@ requires true;
@ assignable \nothing;
@*/
  public /*@ helper @*/ int getCurrentSpeed();
}
```

**Listing 1.1.** The interface*CarControl* and its contracts.

Furthermore, we consider an app `CruiseControl`, which accesses the two methods of the platform's interface `CarControl` as shown in Listing 1.2. For the sake of simplicity, we set a fixed target speed in the `run` method, which incrementally increases the current speed and terminates once the target speed is reached.

```
public class CruiseControl {
  CarControl cc = new CarControlImpl();
/*@
@ requires true;
@ diverges true;
@*/
  public void run(){
    int targetSpeed = 200;
    int currentSpeed = getNormalizedSpeed();
/*@ loop_invariant currentSpeed >= 0 && currentSpeed <= 300;
@ assignable \nothing;
@*/
    while(currentSpeed != targetSpeed){
      if(currentSpeed < targetSpeed){ cc.setTargetSpeed(currentSpeed + 1);}
      else { cc.setTargetSpeed(currentSpeed − 1); }
      currentSpeed = getNormalizedSpeed();
    }}
/*@
@ requires true;
@ ensures \result >= 0 && \result <= 300;
@ assignable \nothing;
@*/
  public int getNormalizedSpeed() {
    int currentSpeed = cc.getCurrentSpeed();
    if(currentSpeed > 300) { currentSpeed = 300; }
    else if(currentSpeed < 0) { currentSpeed = 0; }
    return currentSpeed;
}}
```

**Listing 1.2.** The app *CruiseControl* and its contracts.

## 5.2 Proof-Carrying Apps with KeY

As a first instance of proof-carrying apps, we provide an implementation utilizing the KeY tool for deductive program verification [1]. We give an introduction to the concept and determine whether KeY supports the speed-up of a proof check necessary for proof-carrying apps.

*Approach* The KeY approach uses deductive reasoning as a main technology. It allows verifying Java programs using symbolic execution. At certain decisions in the proof tree, it might be necessary to manually choose a deduction rule or a specific instantiation of a rule. KeY allows storing the proof of a verification pass, which is essentially a proof script. It includes all rules (and how they were instantiated) necessary to deduce the proof and the order in which the rules have to be applied. This proof can then be reopened and used to check whether it holds for the contracts of the program. As the rules only have to be applied and not to be found, we assume that the replay is faster than finding the rules. This is especially true if the automatic mode encounters a point where manual interaction is necessary. KeY also does not need to trust the generated proof script, as all rules in the proof script are only applied if allowed. If the proof and the program are not compatible, KeY cannot finish the proof and the plug-in is rejected. KeY fulfills the three criteria *fully automatically*, *reduced verification effort*, and *no trust in artifacts necessary* introduced in Section 3.

*Adapting the case study* To get the case study to work with KeY, we had to add some additional contracts. We assume that the implementation of the `CarControl` works correctly and only check whether the precondition is not violated by the new plug-in. Therefore, we use the `helper` modifier and tell KeY that the methods do not change the state of the object with the `assignable` clause. KeY only allows to verify contracted methods. To prove that methods using the interface obey the contract, app developers have to add contracts to their own source code. This can be done by simply adding a precondition which always holds, using `requires true`. The methods `run()` and `getNormalizedSpeed()` have a `requires true` precondition and can therefore be verified with KeY. In the implementation of the `CruiseControl`, the while loop has to be annotated by a loop invariant to allow KeY to verify the method. Without the loop invariant KeY terminates with open goals, i. e., the proof cannot be finished.

If developers want to use the interface `CarControl`, the contracts give preconditions they have to meet to be allowed to run. With this knowledge, developers of an app write their *plug-in* (in this case in the form of simple Java classes). It is their responsibility to ensure that the plug-in maintains the API's contracts, which may include adding necessary contracts to their source code. Developers have to find proofs for all methods with contracts by using KeY, which generates a proof script for every method. These files constitute the proof artifacts for the plug-in. The proof and the source code are then verified using KeY as the proof checker. It reads the proof script and attempts to do a proof replay. If the replay succeeds for all methods of the plug-in, the entire plug-in is verified and can be executed. After starting the KeY system and all rules are found and applied, the proof can be saved for a later replay. Developers end up with Java source-code files and a KeY proof artifact.

*Results* When run fully automatically (*autopilot mode*), KeY was able to prove that the preconditions of the interface `CarControl` are satisfied. To detect a violation of the precondition, we modified the field `targetSpeed`, the contract of `getNormalized-Speed` and the normalization values in `getNormalizedSpeed`. All modifications were detected and no proofs were found. The successful proof was saved and reloaded into KeY, which could correctly replay the proof. Modifying the proof or the sources resulted in an exception or remaining open proof goals.

To quantify the benefit of the proof replay, we measured the time it took for the KeY tool to find the proof and to replay it. The developers provided us with a version of KeY that allows measuring the time for parsing the source code and the proof as well as the time required to find or replay a proof. The system used for the test is an Intel Core i7-5600U @2.6GHz with 12 GB RAM running Windows 10 x64. As input, we used the code of the interface `CarControl` and the class `CruiseControl`, annotated with the contracts in JML. Table 1 shows the average times over five runs. With over five seconds, most of the time is used by parsing the source code, which is the same for finding and replaying the proof. In addition to parsing the source code, the proof replay makes it necessary to parse the proof. The time it took to find the proof is significantly larger than the time it took to replay the same proof. Considering the parsing time, the benefit decreases to only 3.4% for the presented example. Realistic systems would be more complex and use more interfaces with preconditions. Due to this reason, we expect the impact of the parsing time for the proof to decrease. The

| **Average Time (5 passes)** | | | |
|---|---|---|---|
| **Proof Search in Autopilot Mode** | | **Proof Replay** | |
| Parsing of Sources | 5150 ms | | Parsing of Sources |
| | | 1558 ms | Parsing of Proof |
| Proof Search | 2882 ms | 1050 ms | Proof Replay |
| Sum | 8032 ms | 7758 ms | Sum |

**Table 1.** The results of the evaluation of our example.

parsing time of the sources and the proof, as well as the proof replay time, should only increase slightly, whereas the time it takes to find a proof is expected to grow faster. Thus, the benefit of replaying a proof should increase significantly.

To achieve a similar effect of detecting and handling unspecified behavior defensive programming could be used. Every method with a precondition has to check its input values whether they abide the contract. If this is not the case, the call can be ignored or error handling can be initiated. This allows detecting and handling, but not the prevention of errors. Defensive programming makes the callee responsible for error detection, although one reason to use contracts is to make the caller responsible for abiding preconditions of the callee. Especially for time critical systems handling erroneous behavior might not be possible because the handling takes time. Additionally the comfort of the driver might be affected if the car pulls over and stops due to an error in the software which could have been prevented in the first place.

## 5.3 Proof-Carrying Apps with Soot

As a second instance of proof-carrying apps, we use Soot, an optimization framework for Java source and byte code. Soot also allows to run static data-flow analyses. After a short introduction we show how we adapted the case study and what results we received.

*Approach* Soot translates the Java code (also Android code and Java bytecode) to an internal representation, which then can be analyzed with Soot. It provides several built-in analyses, as call-graph analyses, points-to analyses, and def-use-chains. Moreover, a template driven intra-procedural data-flow analysis framework is provided that may be used to extend Soot with own analyses. Arzt et al. have introduced a tool called *Reviser* [4]. It is built on top of Soot and allows to update results of data-flow analyses due to program changes without a complete recalculation of the result. Initially, Reviser consumes about 20% more computation time for the analyses. However, the re-computation time decreases by up to 80%. Reviser generates an inter-procedural control-flow graph which is used for the analysis. After the first analysis is finished, the second generation does not create a new graph, but it only changes the nodes for which the source changed. If a node is changed, the change is propagated through all dependent nodes. With the changes of the control-flow graph, the analysis result of the previous pass is updated. A recalculation is only done where necessary. This means Soot/Reviser supports the two criteria to *run fully* *automatically* and to *reduce the verification effort* (see Section 3). For our experiments, we used the version of Soot provided for the *Reviser*[3].

*Adapting the case study* While adapting the *CruiseControl* use case to be verifiable by Soot/Reviser, we encountered several issues. If Reviser does *not* find changes from one version of a program to another, it assumes the results of the first run are still valid without rechecking them. For proof-carrying apps, it means that the proof artifact delivered with the plug-in is not checked at all. Hence, Reviser does not fulfill our criterion to *not trust artifacts*. For the case that Reviser does not find changes, we can enforce an update of the analysis. To ensure we get the correct results, the analysis has to be redone completely. This would potentially increase the verification time, due to the time it takes to parse the artifacts delivered with the plug-ins. To work with our two scenarios Soot/Reviser has to be changed to not trust the provided artifacts. For our encoding of contracts as data-flow problems we rely on potentially complex conditional expressions. The implemented analyses in Reviser abstract heavily from if-else-conditions and combines different paths in the control-flow graph. Every contract would be reported as broken and we would get many false positives, essentially leading to no app being allowed to run.

With the necessity to use trusted artifacts, Soot/Reviser is not suitable for our current use case of proof-carrying apps. Rather than using it to verify a *newly* deployed app, it might be suitable for the different use case of *updates* of apps. In this use case a first analysis would be done without an proof artifact and the result would be saved on the device. If the app is updated, Reviser can calculate the differences between the old and the new app, and update the result of the last analysis. We will investigate this use case in future work.

## 5.4 Summary

Both KeY and Soot were not created for our scenario of proof-carrying apps. In particular, proof replay and parsing of KeY are not optimized for performance. For our very small use case, the verification effort with KeY could be reduced by 63.6%. With the added time for parsing the proof, the resulting reduction is 3.4%. The results show that making proof replays faster, e. g., by optimizing the parsing phase, needs to be addressed in future work.

The current implementation of Soot/Reviser is not suitable for proof-carrying apps, due to the fact that it trusts the provided artifacts. This might change if we extend our scenarios to include update scenarios or the implementation is changed to distrust artifacts. Java PathFinder fulfills our criterion of being able to verify fully automatically. Whereas we found approaches to reduce the verification effort of Java PathFinder [17, 25], we did not find publicly available implementations to evaluate if these approaches suffice our criteria.

[3] https://github.com/StevenArzt/

## 6 Related Work

Our approach extends on proof-carrying code and introduces contracts to specify program properties to prove.

*Proof-Carrying Code* Proof-Carrying Code (PCC) was introduced by Necula to ensure that a platform running code from untrusted developers can be executed in a safe manner [22]. The focus of PCC are global safety rules and, at the beginning, the system was limited to the DEC Alpha assembly language. The source code of a program is compiled and a certificate is created which proves that the safety rules are met. After compilation, the binary and the according proof are deployed. On the platform, the proof is validated and, if it holds, the program is allowed to run. Necula et al. extended a certifying C compiler that automatically generates DEC Alpha assembly and a formal proof that the safety rules hold [23]. The idea of a certifying compiler was extended to support Java by Colby et al. [11]. The MOBIUS project also applied the idea of proof-carrying code for Java. As a target platform, mobile phones were selected which were able to run midlets, i. e., applications implemented using the MIDP profile for JavaME CLDC. The checked properties include points-to and alias information, data dependency, flow of string values, and resource-oriented analysis [6].

*Contracts* Several languages exist which allow specifying permissible use of interfaces. Hatcliff et al. provide an extensive survey on different languages for behavioral interface specifications [13]. For the Java programming language, the Java Modeling Language (JML) is commonly used. It allows specifying preconditions, postconditions, invariants, and assertions, which may be written directly into the Java source code as comments. As KeY natively works with JML and the program properties we are interested in may be expressed as preconditions, we decided to use JML for our case study.

*Incremental and Evolutionary Verification* Tools and methodologies for incremental static analyses already exist. The *Java PathFinder* was intended as model checker for Java programs. It translates Java code to a *Promela* model, which may be analyzed [14]. Thus, it is possible to find all paths through the program. Moreover, it provides good traceability via backtracking. JPF is modular and, therefore, easy to extend. Brat et al. have shown that JPF is suitable for static analyses [9]. Additionally, Lauterburg et al. [17] as well as Person et al. [25] have extended JPF with incremental analyses. Lauterburg et al. could measure performance increases from 14% up to 68% compared to non-incremental analyses. However, the values strongly depend on the changes made to the program and the overall lines of code.

The Saturn program analysis system[4] translates programs written in C to boolean constraints [2]. These constraints can be extended by boolean expressions for custom analyses. Saturn is intended for static program analyses and provides two different analyses methods. The *whole-program* analysis translates the complete source code of the program to boolean constraints. In contrast, the *compositional-summary-based* analysis translates every function individually to boolean constraints. Afterwards,

[4] http://saturn.stanford.edu

these constraints can be composed to represent the entire program or particular parts of the program. Incremental analysis could be realized by only analyzing functions which are affected by evolution. However, this is a relatively coarse-grained method. Mudduluru et al. [21] have extended Saturn to *incremental Saturn* (iSaturn). iSaturn is supposed to provide more fine-grained increments. For this, the boolean constraints are analyzed for (partial) equivalence. Thus, existing results can be reused, which results in performance increases by up to 32%.

The tool *KeY Resources*[5] allows to automatically find proofs in the background. It is also able to reduce the effort to *find* a proof after the source code or the contracts are changed. The introduction of abstract contracts allows the reuse of parts of a proof *found* by KeY [12]. The authors extended the approach to fully abstract contracts and included abstract invariants [10]. Abstract contracts allowing a developer to reduce the effort to *find* a proof, but for the moment it is unclear how these approaches can help us with *checking* proofs.

## 7 Conclusion and Future Work

In this paper, we introduced proof-carrying apps as a generalization to proof-carrying code. Whereas proof-carrying code uses global safety policies, we use contracts to specify the assumptions developers have towards apps using their API. Instead of only relying on deductive proofs, we allow the use of other proof artifacts that enable the faster verification of the contracts of a plug-in. Contracts have the additional benefit of explicitly splitting the responsibility between the calling and the called object. Furthermore, we describe three criteria a verification technique has to fulfill to work for proof-carrying apps: *fully automatic verification*, *generation of proof artifacts without the need of trust*, and *a decreased verification effort due to the artifact*. We determined that data-flow analysis, software-model checking, and deductive program verification are in principle suitable for proof-carrying apps. We evaluated implementations of the verification techniques, such as KeY and Soot, and found potential of reusing existing techniques for proof-carrying apps but also discuss further requirements for those tools.

For future work, it is necessary to focus on larger case studies as that discussed above and also on other languages than Java. For instance, analysis tools like FRAMA-C[6], VCC[7], or CPAchecker provide tool support for C, which could be interesting to consider for proof-carrying apps. Furthermore, we want to investigate how a speed-up in replaying proofs with KeY can be achieved.

## Acknowledgements

We thank Christoph Seidl for his support in earlier phases of the paper. We gratefully acknowledge the support of Eric Bodden and Steven Arzt with Soot and Reviser.

[5] http://www.key-project.org/eclipse/KeYResources/index.html
[6] http://frama-c.com/index.html
[7] http://vcc.codeplex.com/

## References

1. Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P.H., Ulbrich, M.: The KeY Platform for Verification and Analysis of Java Programs. In: Giannakopoulou, D., Kroening, D. (eds.) Verified Software: Theories, Tools and Experiments. Lecture Notes in Computer Science, vol. 8471, pp. 55–71. Springer (2014), http://dx.doi.org/10.1007/978-3-319-12154-3_4
2. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An Overview of the Saturn Project. In: Workshop on Program Analysis for Software Tools and Engineering. pp. 43–48. PASTE '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1251535.1251543
3. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, Berlin, Heidelberg (2013)
4. Arzt, S., Bodden, E.: Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes. In: International Conference on Software Engineering. pp. 288–298. ICSE 2014, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2568225.2568243
5. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
6. Barthe, G., Crégut, P., Grégoire, B., Jensen, T.P., Pichardie, D.: The MOBIUS Proof Carrying Code Infrastructure. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects. Lecture Notes in Computer Science, vol. 5382, pp. 1–24. Springer (2007), http://dx.doi.org/10.1007/978-3-540-92188-2_1
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004), http://dx.doi.org/10.1007/978-3-662-07964-5
8. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Computer Aided Verification. pp. 184–190 (2011), http://dx.doi.org/10.1007/978-3-642-22110-1_16
9. Brat, G., Visser, W.: Combining Static Analysis and Model Checking for Software Analysis. In: International Conference on Automated Software Engineering. pp. 262–. ASE '01, IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=872023.872568
10. Bubel, R., Hähnle, R., Pelevina, M.: Fully abstract operation contracts. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8803, pp. 120–134. Springer (2014), http://dx.doi.org/10.1007/978-3-662-45231-8_9
11. Colby, C., Lee, P., Necula, G.C., Blau, F., Plesko, M., Cline, K.: A certifying compiler for Java. In: Lam, M.S. (ed.) Conference on Programming Language Design and Implementation. pp. 95–107. ACM (2000), http://doi.acm.org/10.1145/349299.349315
12. Hähnle, R., Schaefer, I., Bubel, R.: Reuse in software verification by abstract method calls. In: Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. pp. 300–314 (2013), http://dx.doi.org/10.1007/978-3-642-38574-2_21
13. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.J.: Behavioral Interface Specification Languages. ACM Comput. Surv. 44(3), 16 (2012), http://doi.acm.org/10.1145/2187671.2187678

14. Havelund, K., Pressburger, T.: Model checking JAVA programs using JAVA PathFinder. International Journal on Software Tools for Technology Transfer 2(4), 366–381 (2000), http://dx.doi.org/10.1007/s100090050043
15. Jhala, R., Majumdar, R.: Software Model Checking. ACM Computing Surveys (CSUR) 41(4), 21 (2009)
16. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The Soot Framework for Java Program analysis: a Retrospective (2011)
17. Lauterburg, S., Sobeih, A., Marinov, D., Viswanathan, M.: Incremental State-space Exploration for Programs with Dynamically Allocated Data. In: International Conference on Software Engineering. pp. 291–300. ICSE '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1368088.1368128
18. Leavens, G.T.: JML: Expressive Contracts, Specification Inheritance, and Behavioral Subtyping. In: Principles and Practices of Programming on The Java Platform. p. 1 (2015), http://doi.acm.org/10.1145/2807426.2817926
19. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370 (2010), http://dx.doi.org/10.1007/978-3-642-17511-4_20
20. Meyer, B.: Applying "Design by Contract". IEEE Computer 25(10), 40–51 (1992), http://doi.ieeecomputersociety.org/10.1109/2.161279
21. Mudduluru, R., Ramanathan, M.: Efficient Incremental Static Analysis Using Path Abstraction. In: Gnesi, S., Rensink, A. (eds.) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol. 8411, pp. 125–139. Springer Berlin Heidelberg (2014), http://dx.doi.org/10.1007/978-3-642-54804-8_9
22. Necula, G.C.: Proof-Carrying Code. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Symposium on Principles of Programming Languages. pp. 106–119. ACM Press (1997), http://doi.acm.org/10.1145/263699.263712
23. Necula, G.C., Lee, P.: The Design and Implementation of a Certifying Compiler. In: Davidson, J.W., Cooper, K.D., Berman, A.M. (eds.) Conference on Programming Language Design and Implementation. pp. 333–344. ACM (1998), http://doi.acm.org/10.1145/277650.277752
24. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999), http://dx.doi.org/10.1007/978-3-662-03811-6
25. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed Incremental Symbolic Execution. In: Conference on Programming Language Design and Implementation. pp. 504–515. PLDI '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1993498.1993558
26. Rushby, J.M.: Design and Verification of Secure Systems. SIGOPS Oper. Syst. Rev. 15(5), 12–21 (Dec 1981), http://doi.acm.org/10.1145/1067627.806586
27. Schumann, J.M.: Automated Theorem Proving in Software Engineering. Springer-Verlag (2001)
28. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Comput. Surv. 47(1), 6:1–6:45 (2014), http://doi.acm.org/10.1145/2580950