Programs from Proofs – A PCC Alternative[⋆]

Daniel Wonisch, Alexander Schremmer, and Heike Wehrheim

University of Paderborn
Germany
{alexander.schremmer,wehrheim}@upb.de

Abstract. Proof-carrying code approaches aim at safe execution of untrusted code by having the code producer attach a safety proof to the code which the code consumer only has to validate. Depending on the type of safety property, proofs can however become quite large and their validation - though faster than their construction - still time consuming.

In this paper we introduce a new concept for safe execution of untrusted code. It keeps the idea of putting the time consuming part of proving on the side of the code producer, however, attaches no proofs to code anymore but instead uses the proof to *transform* the program into an *equivalent* but *more efficiently verifiable* program. Code consumers thus still do proving themselves, however, on a computationally inexpensive level only. Experimental results show that the proof effort can be reduced by several orders of magnitude, both with respect to time and space.

1 Introduction

Proof-Carrying Code (PCC) has been invented in the nineties of the last century by Necula and Lee [21,22]. It aims at the safe execution of untrusted code: (untrusted) code producers write code, ship it via (untrusted) mediums to code consumers (e.g., mobile devices) and need a way of ensuring the safety of their code, and in particular a way of convincing the consumers of it. To this end, code producers attach safety proofs to their code, and consumers validate these proofs. This approach is *tamper-proof*: malicious modifications of the code as well as of the proof get detected.

This general framework has been instantiated with a number of techniques, differing in the type of safety property considered, proof method employed and proof validation concept applied (e.g. [10,3,13]). However, there are also a number of disadvantages associated with PCC which have hindered its widespread application. First of all, we need a specific PCC instance for the type of safety properties our consumer is interested in. Looking at the large spectrum of properties covered, this might turn out to be a solvable problem. However, the instance that we find might only produce very large proofs (which need to be attached to the code) or might only have expensive (wrt. time and space) proof checking techniques.

[⋆] This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

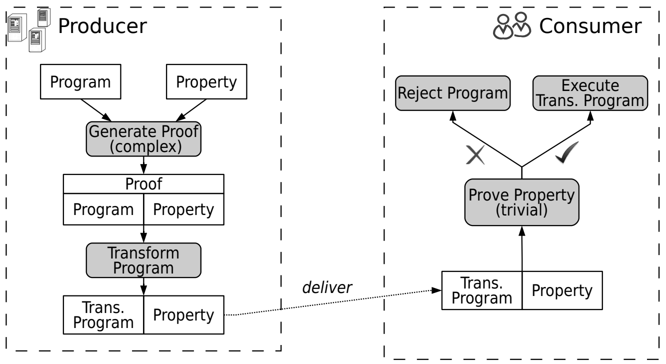

Fig. 1. Overview of our approach

In this paper we propose a different concept for the safe execution of untrusted code. Like PCC it is first of all a general concept with lots of possible instantiations, one of which will be introduced here. Our concept keeps the general idea behind PCC: the potentially untrusted code producer gets the major burden in the task of ensuring safety while the consumer has to execute a time and space efficient procedure only. Figure 1 gives a general overview of our approach: the producer carries out a proof of correctness of the program with respect to a safety property. The information gathered in the proof is next used to *transform* the program into an *equivalent*, more efficiently verifiable (but usually larger wrt. lines of code) program. The transformed program is delivered to the consumer who – prior to execution – is also proving correctness of the program, however, with a significantly reduced effort. The approach remains tamper-proof since the consumer is actually verifying correctness of the delivered program.

Key to the applicability of this concept is of course the possibility of finding instances of proof techniques, property classes and transformations actually exhibiting these characteristics. We are confident that a number of such instances can be found, and present one such instance as a proof of concept here. Our instantiation uses *software model checking* as proof technique on the producer side and *data flow analysis* on the consumer side.

More detailedly, the instance we present here allows for safety-property definitions in terms of *protocol* (or trace) specifications on function calls (or more general, program operations). The producer then employs a *predicate analysis* [16,1,4] on the program to prove properties. The predicate analysis constructs an abstract reachability tree (ART) giving us information about the possible execution paths of the program. The transformation uses this information to construct an equivalent program which has a control-flow graph (CFG) isomorphic to the ART. Original and transformed program are not only equivalent with respect to the behaviour but also with respect to performance (optimisations on the transformed program might even give us a program with less runtime). On the CFG of the transformed program the consumer can easily validate the absence of errors using a simple data flow analysis which is linear in the size of the CFG (no iterative fixpoint computation needed).

In order to evaluate our approach, we have implemented it using the software verification tool CPAchecker [8] for predicate analysis and the tool Eli [17] to implement the simple data flow analysis. Evaluation on some standard benchmark C programs shows that the proof effort for the consumer is significantly lower than for the producer, even though the transformed program is usually larger (in terms of lines of code) than the original program.

2 Preliminaries

For the presentation in this paper, we consider programs to be written in a simple imperative single-threaded programming language with assignments, assume operations and function calls as only possible statements, and variables that range over integers only. The programs considered in the experimental results (Section 5) are however not restricted this way. The left of Figure 2 shows our running example of a program which is calling some lock and unlock functions. The objective is to show that this program adheres to common locking idioms (which it does), i.e., in particular no unlock can occur before a lock.

Formally, a *program* $P = (A, l_0)$ is represented as *control-flow automaton* (CFA) A together with a start location l_0. A CFA $A = (L, G)$ consists of a set of (program) locations L and a set of edges $G \subseteq L \times Ops \times L$ that describe possible transitions from one program location to another by executing certain operations Ops. A *concrete data state* $c : X \to \mathbb{Z}$ of a program P is a mapping from the set of variables X of the program to integer values. The set of all concrete data states in a program P is denoted by $\mathscr{C}$. A set of concrete data states can be described by a first-order predicate logic formula φ over the program variables (which we make use of during predicate abstraction). We write $[\![\varphi]\!] := \{c \in \mathscr{C} \mid c \models \varphi\}$ for the set of concrete data states represented by some formula φ. Furthermore, we write $\gamma(c)$ for the representation of a concrete data state as formula (i.e. $[\![\gamma(c)]\!] = \{c\}$). Note that we assume the program to be started in some arbitrary data state c_0.

A tuple (l, c) of a location and a concrete data state of a program is called *concrete state*. The *concrete semantics* of an operation $op \in Ops$ is defined in terms of the *strongest postcondition operator* $SP_{op}(\cdot)$. Intuitively, the strongest postcondition operator $SP_{op}(\varphi)$ of a formula φ wrt. to an operation op is the strongest formula ψ which represents all states which can be reached by op from a state satisfying φ. Formally, we have $SP_{x:=expr}(\varphi) = \exists \widehat{x} : \varphi_{[x \mapsto \widehat{x}]} \wedge (x = expr_{[x \mapsto \widehat{x}]})$ for an assignment operation $x := expr$, $SP_{assume(p)}(\varphi) = \varphi \wedge p$ for an assume operation $assume(p)$ ($assume(\cdot)$ omitted in figures) and $SP_{f()}(\varphi) = \varphi$ for a function call $f()$. Thus, we assume function calls to not change the data state of our program. Our implementation lifts this limitation, which was done only for presentation purposes.

```
init();
lock();
int lastLock = 0;
for(int i = 1; i < n; i++) {
  if( i - lastLock == 2 ) {
    lock();
    lastLock = i;
  } else {
    unlock();
   }
}
```

```
init();
lock();
int lastLock = 0;
for(int i = 1; i < n; i++) {
  unlock();
  i++;
  if( i >=n ) { break; }
  lock();
  lastLock = i;
}
```

Fig. 2. Program LOCKS (left) with its transformation to an equivalent program (right)

We write $(l, c) \xrightarrow{g} (l', c')$ for concrete states (l, c), (l', c') and edge $g := (l, op, l')$, if $c' \in [\![SP_{op}(\gamma(c))]\!]$. We write $(l, c) \rightarrow (l', c')$ if there is an edge $g = (l, op, l')$ such that $(l, c) \xrightarrow{g} (l', c')$. The feasible *paths* of a program $P = (A, l_0)$ with CFA $A = (L, G)$ are the sequence of concrete states the program can pass through:

$$paths(P) := \{(c_0, \dots, c_n) \mid \exists l_1, \dots, l_n \in L, \exists g_1, \dots, g_{n-1} \in G : \\ (l_0, c_0) \xrightarrow{g_1} \dots \xrightarrow{g_{n-1}} (l_n, c_n)\}$$

Similarly, the set of *traces* of a program P are the sequences of operations it can perform:

$$traces(P) := \{(op_0 \dots op_n \mid \exists l_1, \dots, l_n \in L, \exists c_0, \dots, c_n \in \mathscr{C}, \exists g_1, \dots, g_{n-1} \in G : \\ (l_0, c_0) \xrightarrow{g_1} \dots \xrightarrow{g_{n-1}} (l_n, c_n) \wedge \forall 0 \leq i < n : g_i = (l_i, op_i, l_{i+1})\}$$

We are ultimately interested in proving *safety* properties of programs. Safety properties are given in terms of protocol automata which describe the allowed sequences of operations.

Definition 1. *A* protocol or property automaton $A_{prop} = (\Sigma, S, s_0, s_{err}, \delta)$ *consists of an alphabet Σ, a finite set of states S with initial state s_0 and error state s_{err}, and transition relation $\delta \subseteq S \times \Sigma \times S$. The transition relation is deterministic. The error state has outgoing transitions $(s_{err}, op, s_{err}) \in \delta$ for all $op \in \Sigma$.*

The language $L(A_{prop})$ *of a protocol automaton is the set of traces $op_1 \dots op_n$ such that $\delta^*(s_0, op_1 \dots op_n) \neq s_{err}$.*

The property automaton in Figure 3 describes all valid locking patterns: first, a call of *init*() needs to be performed and then *lock*() and *unlock*() have to occur in turns. The operations occuring in property automata are usually function calls, however, these can also be any syntactic program property that is expressible as a BLAST automaton [5].

The property automaton only speaks about a part of the program operations, namely those in Σ. A comparison of program and protocol automaton traces thus

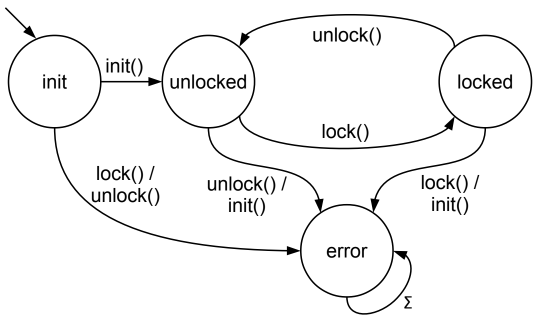

Fig. 3. Property Automaton. Disallows two lock() or unlock() in a row.

needs to project the traces of the program onto the alphabet of the automaton (projection written as $\upharpoonright$). Hence, program P *satisfies* the safety property of protocol automaton A_{prop}, $P \models A_{prop}$, if $traces(P) \upharpoonright \Sigma \subseteq L(A_{prop})$.

To analyse whether a safety property holds for a program, we use predicate abstraction (as supplied by CPAchecker [8]). CPAchecker builds the product of the property automaton and an abstraction of the concrete state space yielding an *abstract reachability tree* (ART). More precisely, the concrete data states are abstracted by quantifier-free first order predicate logic formulas over the variables of the program. We let PS denote the set of all such formulas. We only sketch the algorithm behind ART construction, for details see for instance [8]. What is important for us, is its form. The nodes of the ART take the form (l, s, φ) describing the location the program is in, the current state of the property automaton and a predicate formula as an abstraction of the data state. We assume that the ART is finite. This is the case if predicate refinement terminates for a given program.

Definition 2. *An* abstract reachability tree $T = (N, G, C)$ *consists of a set of nodes* $N \subseteq L \times S \times PS$*, a set of edges* $G \subseteq N \times Ops \times N$ *and a* covering $C : N \to N$.

The covering is used to stop exploration of the abstract state space at a particular node once we find that the node is covered by an already existing one: if $C(l, s, \varphi) = (l', s', \varphi')$ then $l = l'$, $s = s'$ and $[\![\varphi]\!] \subseteq [\![\varphi']\!]$. The successor of an already constructed node n is constructed by searching for successor nodes in the CFA, computing the abstract post operation on the predicate formula of n and determining the successor property automaton state. After generation of a new ART node, the algorithm checks whether the new ART node is covered by an existing one and generates an entry in the covering if necessary. One result of this process is that loops are unrolled such that within ART nodes program locations are only associated to a *single* state of the property automaton: if a program when reaching location l can potentially be in more than one state of the (concurrently running) protocol automaton, these will become separate nodes in the ART.

Figure 4 shows the ART of program LOCKS as constructed by CPAchecker. The dotted line depicts the covering. The ART furthermore satisfies some healthiness conditions which we need further on for the correctness of our construction (and which is guaranteed by the tool we use for ART construction):

Soundness. If $((l, s, \varphi), op, (l', s', \varphi')) \in G_{art}$, then for all $c \in [\![\varphi]\!]$ with $(l, c) \overset{(l,op,l')}{\rightarrow} (l', c')$ we have $c' \in [\![\varphi']\!]$. Furthermore, if $op \in \Sigma$, then $(s, op, s') \in \delta_{A_{prop}}$, and $s = s'$ else.

Completeness. If (l, op, l') is an edge in the CFG of the program, then for all $(l, s, \varphi) \in N \setminus \mathrm{dom}(C)$ with $\varphi \not\equiv false$, we have some $(l', \cdot, \cdot) \in N$ such that $((l, s, \varphi), op, (l', \cdot, \cdot)) \in G_{art}$.

Determinism. For every $n \in N$, there is only one successor node in G_{art} except when nodes have outgoing assume edges. In this case, also more than one successor nodes of n are allowed.

Well-Constructedness. For every $((l, \cdot, \cdot), op, (l', \cdot, \cdot)) \in G_{art}$, we have an edge (l, op, l') in the CFG. For $P = (A, l_0)$ and root of the ART (l_0, s_0, φ_0), we have $l = l_0$, $s = s_0$ and $[\![\varphi_0]\!] = \mathscr{C}$.

As a consequence, the program satisfies the property of the protocol automaton if the ART does not contain a node (l, s, φ) with $s = s_{err}$. Note that it is not always possible to construct a finite ART. In general, ART construction consists of incremental abstraction, checking and refinement steps (CEGAR) until the property is proven. However, our interest here is not in techniques for the initial proof of correctness by the producer of a program, but in a method for simplifying subsequent proofs by consumers. Thus we assume in the following that the program satisfies the property, and that we have an ART with the above mentioned characteristics available.

3 Program Transformation

The construction of the abstract reachability tree is the task of the producer. He or she needs to show that the program adheres to the safety property and needs a way of convincing the producer of this fact. Proving can be time-consuming, and this is an accepted property of PCC techniques. Proof checking on the consumer side should however be easy.

The first step of our method is now the transformation of the program into another program for which *proving* is much easier. So instead of "easy proof checking", in our approach the consumer is carrying out "easy proving".

The transformation proceeds by constructing – in its most basic form – a goto program from the ART. Later optimisations bring this into a more readable form, but here we just formally define transformation into goto programs by giving the new program again in the form of a control flow automaton and an initial location. The idea is quite simple: every node in the ART becomes a location in the new program, and the operations executed when going from one node to the next are those on the edges in the ART.

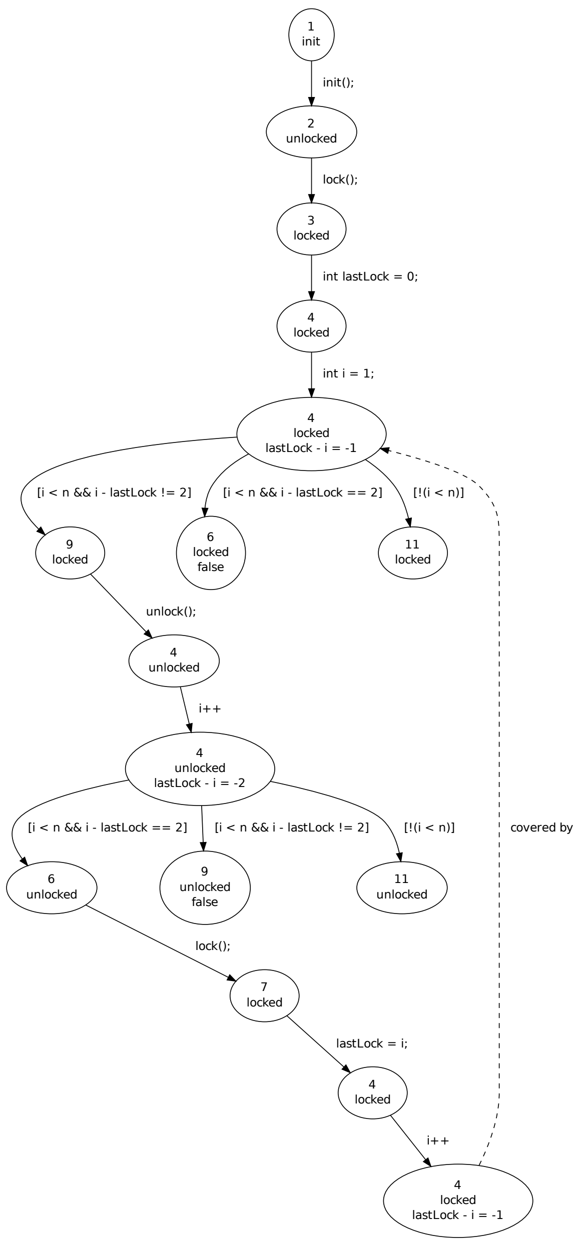

Fig. 4. Abstract reachability tree as generated by CPAchecker. Not all states bear abstraction predicates as the predicate analysis performs the abstraction only at loop heads in adjustable-block encoding mode [4].

Definition 3. *Let $T = (N, G, C)$ be an abstract reachability tree generated from a program and a property automaton. The* transformed program*, $program\_of(T)$, is a program $P' = ((L', G'), l'_0)$ with $L' = N \backslash dom(C)$, $l'_0 = root(T) = (l_0, s_0, true)$ and edges defined as*

$$(l_1, op, l_2) \in G' \Leftrightarrow \begin{cases} (l_1, op, l_2) \in G & \text{if } l_2 \notin dom(C) \\ (l_1, op, l_3) \in G \wedge (l_3, l_2) \in C & \text{otherwise.} \end{cases}$$

This is well defined because $\mathrm{dom}(G) \cap \mathrm{dom}(C) = \emptyset$, $C(N) \cap dom(C) = \emptyset$, and since the ART is deterministic. This representation can be easily brought back into a programming languge notation using gotos, and with some effort into a program without gotos and proper loops instead (assuming the resulting loop structure is reducible). The right of Figure 2 shows the transformed version of program LOCKS in the form which uses loops. Note that the transformation of a program containing gotos to the version containing loops is solely done for presentation purposes.

Due to the fact that the edges in the ART and thus the statements in the new program are exactly those of the original program, we obtain a new program which is equivalent to the original one: it possesses the same paths.

Theorem 1. *Let P be a program. Let $ART = (N, G_{art}, C)$ be an abstract reachablity tree for P. Let $P' = program\_of(ART)$. Then $paths(P) = paths(P')$.*

Furthermore, we perform a small optimisation that does not affect the correctness of the transformation. We also omit all edges in the ART leading to nodes labelled false as these represent steps the program will never execute. Furthermore, if after this removal there is just one outgoing edge from a node and this edge is labelled with an assume operation, we delete this assume.

The obtained program has exactly the same behaviour as the original program and thus the desired functionality. The lines of code usually increase during transformation. The performance (run time) however stays the same or even decreases. This is the case when the optimisation removes assume operations on edges (because the ART shows that this condition always holds at the particular node). For program LOCKS we see that the loop has been unfolded once, and the test for equality (`i-lastLock == 2`) could be removed.

4 Program Validation

The transformed program is given to the consumer and ready for use. The consumer wants to ensure that the program he/she uses really adheres to the safety property. For this, the consumer can use a computationally inexpensive data flow analysis; inexpensive because – unlike standard DFAs – the information needed can be computed without an iterative fixpoint computation.

Algorithm 1 shows the overall procedure. The objective is here to – again – build a product of program and property automaton, however, this time considering no data states at all. The algorithm just computes for every location of

Algorithm 1. Data-flow analysis as used to validate the conformance of a given program to a given protocol property.

```
Input: Program $P = ((L, G), l_0)$, protocol automaton $A_{prop} = (\Sigma, S, s_0, s_{err}, \delta)$
Output: Mapping $m : L \rightarrow S \cup \{\bot, \top\}$
 1: for all $l \in L$ do
 2:     $m(l) := \bot$;
 3: $m(l_0) := s_0$;
 4: $stack.push(l_0)$;
 5: while $stack$ is not empty do
 6:     $l := stack.pop()$;
 7:     for all $(l, op, l') \in G$ do
 8:         if $op \in \Sigma$ then
 9:             $s' := \delta(m(l), op)$;
10:         else
11:             $s' := m(l)$;
12:         if $m(l') = \bot$ then
13:             $m(l') := s'$;
14:             $stack.push(l')$;
15:         else if $m(l') \neq s'$ then
16:             return $\{l \mapsto \top \mid l \in L\}$;
17: return m;
```

the program the state the property automaton is in when reaching this location. The outcome is a mapping m from locations to states plus the special values $\bot$ (no state assigned to location yet) and $\top$ (more than one state assigned to the location). For arbitrary programs, we could get a *set* of automaton states for a location. However, when the program is the result of our transformation, every location has exactly one associated property automaton state: if the location of the program is derived from ART node (l, s, φ), then $m(l) = s$. Furthermore, given the original program was correct, the state $m(l)$ is never s_{err} (the error state of the property automaton). The algorithm thus determines by a depth-first traversal automaton states for program locations. Once it finds that a second state needs to be added to the states for a location (line 15), it stops.

Lemma 1. *Algorithm 1 has a worst case execution time of* $\mathcal{O}(|L| + |G|)$.

Proof. It is a depth-first search (no fixpoint iteration involved).

The consumer executes this algorithm on the program gained from the producer. If the algorithm terminates with (1) $m(l) \neq \top$ and (2) $m(l) \neq s_{err}$ for all locations l, the program is safe. If (1) does not hold, the program is not the result of our transformation, and either the producer has not followed the transformation scheme or someone else has changed the program after transformation. If (2) does not hold, the producer has given an unsafe program. Both properties are thus detectable. Hence this new approach remains tamper-proof.

We next actually show these properties. The next lemma proves soundness of the algorithm, namely that it computes an overapproximation of the set of states of the property automaton that the program can be in when in a particular location.

Lemma 2 (Soundness). *The mapping* $m : L \to S \cup \{\bot, \top\}$ *returned by Algorithm 1 is a overapproximation of the states the property automaton can be in for each program location. That is, for program* $P = ((L, G), l_0)$ *and protocol automaton* $A_{prop} = (\Sigma, S, s_0, s_{err}, \delta)$*, if* $(l_0, c_0) \overset{op_1}{\to} \ldots \overset{op_n}{\to} (l_n, c_n)$ *then* $\delta^*(s, (op_1, \ldots, op_n) \upharpoonright \Sigma) = m(l_n)$ *or* $m(l_n) = \top$*.*

Proof. Let $(l_0, c_0) \overset{op_1}{\to} \ldots \overset{op_n}{\to} (l_n, c_n)$ be some path in P. Furthermore, assume $\delta^*(s, (op_1, \ldots, op_n) \upharpoonright \Sigma) \neq m(l_n)$. Furthermore, assume that $m(l) = \top$ for all $l \in L$ does not hold (then $m(l) \neq \top$ for all $l \in L$ by construction of the algorithm). Then, there is a first element in the sequence of locations of the path l_i ($i \in \{1, \ldots, n\}$) such that $\delta^*(s, (op_1, \ldots, op_i) \upharpoonright \Sigma) \neq m(l_i)$ and $\delta^*(s, (op_1, \ldots, op_{i-1}) \upharpoonright \Sigma) = m(l_{i-1})$. Since $m(l_{i-1}) \neq \bot$ and $m(l_{i-1}) \neq \top$, there is an iteration of the while-loop in which l_{i-1} is at the top of the stack. Moreover, as all operations op with $(l_{i-1}, op, \cdot) \in G$ are considered in the inner for-loop of the while-loop, the edge $(l_{i-1}, op_i, l_i) \in G$ is also considered. By construction of the algorithm we then get $s' = \delta(m(l_{i-1}), op_i)$ if $op_i \in \Sigma$ or $s' = m(l_{i-1})$ otherwise. In both cases we thus have $s' = \delta^*(s, (op_1, \ldots, op_i) \upharpoonright \Sigma)$. By construction of the algorithm and by the assumption that $m(l) = \top$ for all $l \in L$ does not hold, we get $m(l_i) = s' = \delta^*(s, (op_1, \ldots, op_i) \upharpoonright \Sigma)$ after the respective iteration of the while-loop. This is in contradiction to $m(l_i) \neq \delta^*(s, (op_1, \ldots, op_i) \upharpoonright \Sigma)$ and not $m(l) = \top$ for all $l \in L$, as the value of $m(l)$ for a given location l is never changed in the algorithm once $m(l) \neq \bot$. □

The second lemma shows preciseness of the algorithm: whenever value $\top$ is computed for some location, then there are at least two syntactically feasible paths of the program which reach different states in the property automaton. In addition, a property automaton state s such that $m(l) = s$ can – at least syntactically – be reached.

Lemma 3 (Preciseness). *Let* $m : L \to S \cup \{\bot, \top\}$ *be the mapping returned by Algorithm 1. Let* $l \in L$*. If* $m(l) = \top$ *then there are two syntactically feasible paths* $l_0 \overset{op_1}{\to} \ldots \overset{op_n}{\to} l_n$, $l_0 \overset{op'_1}{\to} \ldots \overset{op'_m}{\to} l'_m$ *with* $l'_m = l_n$ *such that* $\delta^*(s_0, (op_1, \ldots, op_n) \upharpoonright \Sigma) \neq \delta^*(s_0, (op'_1, \ldots, op'_m) \upharpoonright \Sigma)$*. If* $m(l) \in S$*, then there is a syntactically feasible path* $l_0 \overset{op_1}{\to} \ldots \overset{op_n}{\to} l_n$ *such that* $\delta^*(s_0, (op_1, \ldots, op_n)) = m(l)$*.*

Proof. Let $m(l) = \top$. Thus, Algorithm 1 terminated in line 16 (rather than 17). Let l'_{m-1}, l'_m denote the locations l and l', respectively, as considered in the last iteration of the while-loop. By construction, we have a path $l_0 \overset{op_1}{\to} \ldots \overset{op_n}{\to} l_n$ such that $l_n = l'_m$ and $\delta^*(s_0, (op_1, \ldots, op_n) \upharpoonright \Sigma) = m(l_n)$ (where m refers to the mapping in the algorithm prior returning $\{l \mapsto \top \mid l \in L\}$). Similar, there is a path $l_0 \overset{op'_1}{\to} \ldots \overset{op'_m}{\to} l'_m$ such that $\delta^*(s_0, (op'_1, \ldots, op'_m) \upharpoonright \Sigma) = s'$. Because the algorithm returns in line 16, we have $m(l_n) \neq s'$ and thus $\delta^*(s_0, (op_1, \ldots, op_n) \upharpoonright \Sigma) \neq \delta^*(s_0, (op'_1, \ldots, op'_m) \upharpoonright \Sigma)$. □

Finally, together these give us our desired result: if the original program satisfies the property specified in the protocol automaton, then so does the transformed program and we can check for this in time linear in the size of the transformed program.

Theorem 2. *Let P be a program. Let $P' = program\_of(T)$, where $T = (N, G, C)$ is an ART for P wrt. A_{prop}. Let $P \models A_{prop}$. We have:*

(a) $P' \models A_{prop}$,
(b) $P' \models A_{prop}$ is verifiable in $\mathcal{O}(|N| + |G|)$.

Proof. By Theorem 1 we have $traces(P) = traces(P')$, thus $P' \models A_{prop}$ follows directly from $P \models A_{prop}$. To show that $P' \models A_{prop}$ is verifiable in $\mathcal{O}(|N| + |G|)$ we show that Algorithm 1 can prove $P' \models A_{prop}$ and has an runtime bound of $\mathcal{O}(|N|+|G|)$. The latter directly follows by Lemma 1. For the former we first show that every syntactically feasible path P' ending in l' yields the same automaton state s that is also associated to $l' = (l, s, \varphi)$. Let $l'_0 \overset{op_1}{\to} \dots \overset{op_n}{\to} l'_n$ be some syntactically feasible path in P'. Let (l_i, s_i, φ_i) denote the to l'_i associated ART nodes ($i \in \{0, \dots, n\}$). By induction hypothesis we have $\delta^*(s_0, (op_1, \dots, op_{n-1}) \upharpoonright \Sigma) = s_{n-1}$. We have either $(l_{n-1}, op_n, l_n) \in G$ or $(l_{n-1}, op_n, l_{im}) \in G$, $(l_{im}, l_n) \in C$. In both cases, by soundness of the abstractions in the ART, we have $s_n = \delta^*(s_{n-1}, (op_n) \upharpoonright \Sigma)$. Next, by Lemma 3 it follows that Algorithm 1 returns a mapping m with $m(l) \neq \top$ for all $l \in N$ when applied to P' and A_{prop}. By Lemma 3 we also get that $m(l) \neq s_{err}$ for all $l \in L$, as $m(l) = s_{err}$ would require a syntactic path in the ART that leads to the error state which contradicts $P \models A_{prop}$. Finally, by Lemma 2 we get that if $m(l) \in S \setminus \{s_{err}\}$ or $m(l) = \bot$ holds for all locations $l \in E$, then $P' \models A_{prop}$. □

In summary, we have thus obtained a way of transforming a program into an equivalent one, with the same execution time (or less), on which we can however check our property more efficiently.

5 Experimental Results

For the transformed program we have a linear time algorithm for checking adherence to the property. Still, we can of course not be sure that this brings us any improvement over a predicate analysis on the original program since the transformed program is usually larger than the original one. Thus, the purpose of the experiments was mainly to see whether the desired objective of supplying the consumer with a program which is easy to validate is actually met.

We implemented our technique in the program-analysis tool CPAchecker [8]. We chose the Adjustable-Block Encoding [4] predicate analysis. Additionally, we implemented the (consumer-side) validation of the transformed program as a small program written using the compiler construction toolkit Eli [17] that performs parsing and the data-flow analysis described above. All experiments were performed on a 64 bit Ubuntu 12.10 machine with 6 GB RAM, an Intel i7-2620M at 2.7 GHz CPU, and OpenJDK 7u9 (JVM 23.2-b09). The analysis time does not include the startup time of the JVM and CPAchecker itself (around 1-2 s). The memory usage of the DFA program was measured using the tool Valgrind. The amount of operations in the program can be estimated by looking at the lines of code (LOC) because our transformation to C code inserts a newline at least for every statement.

Beside the basic approach described so far, our implementation contains a further optimization to keep the size of the transformed program small. Instead of performing a single predicate analysis and ART generation only, we perform two passes: the first pass generates the product construction of the CFA and the property automaton yielding ART T_1, and the second pass performs a separate predicate analysis yielding ART T_2. In other words, T_1 is the result of performing the predicate analysis with an empty set of predicates. As a last step, we traverse T_1 and remove every (error) node $(l, s_{err}, true)$ ($true$ represents the empty predicate) such that no (l, s_{err}, φ) appears in T_2 for any φ. Since the program is usually correct (unless the producer tries to sell an incorrect program), this usually means that we remove all nodes in T_1 in which the error state of the protocol automaton occurs. T_1 is then subject to the generation of the new program. The correctness of this step is based on the fact that if the predicate analysis proved that the combination of location and protocol error state is not reachable at all, we can deduce that it must also be unreachable in T_1, because T_1 and T_2 are both overapproximating the program behaviour.

Table 1 shows the benchmark results[1] for different kinds of programs and different analysis'. Except for the lock example from above, the programs in Table 1 were taken from the benchmark set of the Software Verification competition 2012[2]. The token ring benchmark was initially a SystemC program passing tokens between different simulated threads that was amended with a scheduler. The ssl_server (named s3_srvr) benchmark is an excerpt from the OpenSSL project mimicking a server-side SSL connection handler. For the s3_srvr program, we derived different protocol automata of varying complexity. The protocol automaton in case of the token ring benchmark simply checks whether every task is started at most once by the scheduler at the same time.

The *predicate analysis* step in the table includes the predicate analysis that would need to be performed by the code consumer to check correctness of the original, untransformed program. Instead, in our approach this step together with the transformation step is ran by the code producer to generate the C representation of the ART. As the transformation step itself is quite fast ($<$ 300 ms), times were omitted in the table. Afterwards, the code consumer can carry out the cheap *DFA analysis* to show that the transformed program obeys the specified protocol property. To undermine our claim that this technique is an

[1] All benchmark files necessary to reproduce the results are available in the public repository of CPAchecker under `https://svn.sosy-lab.org/software/cpachecker/branches/runtime_verification/`

[2] See `http://sv-comp.sosy-lab.org/`

alternative (in fact, a much better alternative) to proof-carrying code approaches, we also implemented a PCC version based on predicate analysis. The PCC technique attaches the proof in form of the ART and its predicates as a certificate to the program. The *PCC analysis* thus needs to check that the ART is a valid ART and an abstraction of the state space of the program. This in particular necessitates a large number of entailment checks carried out by an SMT solver. However, it avoids abstraction refinement and thus is faster than the original prediate analysis. Still, we see that the speed-up obtained by our transformation based approach is much larger than that of the PCC approach.

Table 1. Benchmark results for different programs and properties

Program (Monitor)	Orig.	Predicate Analysis		PCC Checking		Transf.	DFA Checking	
	LOC	Time	Mem.	Time	Mem.	LOC	Time	Mem.
lock2.c	32	0.066s	88MB	0.068s	88 MB	44	0.00s	0.047 MB
token_ring.02.cil.c	596	7.052s	450MB	1.919s	112 MB	3976	0.01s	0.420 MB
token_ring.03.cil.c	724	14.644s	687MB	5.803s	271 MB	13721	0.01s	1.329 MB
token_ring.04.cil.c	846	83.585s	1139MB	14.281s	592 MB	42944	0.04s	4.121 MB
s3_srvr.cil.c (mon1)	861	25.400s	669MB	5.802s	244 MB	5112	0.00s	0.557 MB
s3_srvr.cil.c (mon2)	861	94.393s	646MB	6.476s	282 MB	4267	0.00s	0.471 MB
s3_srvr.cil.c (mon3)	861	10.616s	484MB	2.701s	158 MB	4267	0.00s	0.471 MB
s3_srvr.cil.c (mon4)	861	20.075s	569MB	5.527s	247 MB	4267	0.01s	0.473 MB
s3_srvr.cil.c (mon5)	861	52.755s	848MB	12.593s	540 MB	7623	0.01s	0.794 MB
s3_srvr.cil.c (mon6)	861	7.385s	390MB	2.926s	138 MB	9279	0.01s	0.999 MB

In summary, we have shown that the consumer-side validation step (DFA) is sped up compared to the analysis step by three to four orders of magnitude[3] (except for the small locks example where the predicate analysis is rather trivial). This significantly demonstrates the usefulness of our approach despite the program size increase of about two to three orders of magnitude. We have also compared the compilation time of the transformed C program with that of the original program to see whether the program size brings compilation times into unwanted ranges: while it increases, it never exceeds one second, thus it stays at an acceptable level.

6 Conclusion

In this paper we have proposed an alternative way of providing the safe execution of untrusted code for code consumers, without imposing the overhead of an extensive verification on the consumer. The approach is based on automatic verification via model checking, more precisely predicate analysis, as the first step for a transformation of the program into a more efficiently analysable program, however, with the same behaviour and performance. For the code consumer

[3] An analysis time of 0.00 seconds means non-measurable amount of time.

correctness can then be proven using a simple data flow analysis. Experimental results have shown that – despite the program usually getting larger during transformation – the analysis can still be significantly simplified.

Related Work. Proof carrying code (PCC) techniques often aim at showing memory and type safety of programs. The PCC approach most closest to ours is that of [18,19] who also use predicate analysis as a basis. In contrast to us they follow a classical PCC approach: the producer generates inductive invariants of the program (which show particular properties) and the validation of the consumer consists of actually checking that the invariant is an invariant. As this involves the call of a theorem prover it will in general take more time than a simple data flow analysis. An exact comparison with our approach was not possible as the programs analysed in [18] are currently not publicly available.

Similar to our approach, some techniques convey the fix-point solution of some iterative algorithm (e.g. abstract interpretation) and check this fix-point using a cheap analysis on the consumer side. Instantiations of this concept can be seen for e.g. Java bytecode [23]. Furthermore, to decrease the amount of memory needed for verification, another approach simplifies the Java bytecode before it is passed to the consumer [20]. In both cases, the class of checkable properties does not amount to safety properties like in our case but is limited to simple type properties on the bytecode level.

The idea of using the abstract reachability tree of a program obtained by a predicate analysis is also the idea underlying conditional model checking [6]. While the transformation of an ART into a program is generally envisaged (for the purpose of benchmark generation), the approach focuses on generating conditions for use in further verification runs. Generation of programs from parts of an ART, namely certain counterexamples, is also the basis of the work on *path invariants* [7] which presents an effective way of abstraction refinement.

In the area of model checking, *slicing* is an established technique for reducing the size of program before verification [14]. This transformation is however not generally behaviour preserving, but only property preserving.

A lot of works focus on verifying protocol-like properties of programs. This starts as early as the eighties with the work on typestate analysis [24]. Typestate is a concept which enhances types with information about their state and the operations executable in particular states. Recent approaches have enhanced typestate analysis with ideas of predicate abstraction and abstraction refinement [12,11]. Others use an inconclusive typestate analysis to generate residual monitors to be used in runtime verification [15,9]. This can be seen as a form of partial evaluation. The tool SLAM [2] uses predicate abstraction techniques to analysis C programs with respect to typestate like properties.

Future Work. In the future, we intend to investigate other instantiations of our general framework, first of all for other property classes (e.g. memory safety). We believe that the basic principle of having a model checker do a path-sensitive analysis constructing an ART, which is transformed into a program on which a single-pass data-flow analysis can then prove the desired property, is applicable to a large number of properties since the construction of the ART unfolds the program in such a way that every node in the ART is "unique" wrt. program location and property of interest. We furthermore aim at using this approach for generation of residual monitors, to enhance runtime verification when the property could not be proven on the program. It would furthermore be interesting to investigate whether the source code property of being "easily verifiable" would in some way carry over to the compiled binary.

References

1. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
2. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
3. Bauer, L., Schneider, M.A., Felten, E.W., Appel, A.W.: Access control on the web using proof-carrying authorization. In: Proceedings of the 3rd DARPA Information Survivability Conference and Exposition, DISCEX (2), pp. 117–119 (2003)
4. Beyer, D., Keremoglu, M., Wendler, P.: Predicate Abstraction with Adjustable-Block Encoding. In: FMCAD 2010, pp. 189–197 (2010)
5. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST query language for software verification. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
6. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: a technique to pass information between verifiers. In: Tracz, W., Robillard, M.P., Bultan, T. (eds.) SIGSOFT FSE, p. 57. ACM (2012)
7. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 300–309. ACM (2007)
8. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
9. Bodden, E., Lam, P., Hendren, L.: Clara: A Framework for Partially Evaluating Finite-State Runtime Monitors Ahead of Time. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 183–197. Springer, Heidelberg (2010)
10. Crary, K., Weirich, S.: Resource bound certification. In: Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 184–198 (2000)
11. Das, M., Lerner, S., Seigle, M.: ESP: Path-Sensitive Program Verification in Polynomial Time. In: PLDI, pp. 57–68 (2002)
12. Dhurjati, D., Das, M., Yang, Y.: Path-sensitive dataflow analysis with iterative refinement. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 425–442. Springer, Heidelberg (2006)
13. Drzevitzky, S., Kastens, U., Platzner, M.: Proof-carrying hardware: Towards runtime verification of reconfigurable modules. In: Proceedings of the International Conference on Reconfigurable Computing (ReConFig), pp. 189–194. IEEE (2009)
14. Dwyer, M.B., Hatcliff, J., Hoosier, M., Ranganath, V.P., Robby, Wallentine, T.: Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 73–89. Springer, Heidelberg (2006)

15. Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In: ASE, pp. 124–133. ACM (2007)
16. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
17. Gray, R., Levi, S., Heuring, V., Sloane, A., Waite, W.: Eli: A complete, flexible compiler construction system. Communications of the ACM 35(2), 121–130 (1992)
18. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 526–538. Springer, Heidelberg (2002)
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 332–358. Springer, Heidelberg (2004)
20. Leroy, X.: Bytecode verification on java smart cards. Softw. Pract. Exper. 32(4), 319–340 (2002)
21. Necula, G.C.: Proof-carrying code. In: POPL 1997, pp. 106–119. ACM, New York (1997)
22. Necula, G.C., Lee, P.: Safe, untrusted agents using proof-carrying code. In: Vigna, G. (ed.) Mobile Agents and Security. LNCS, vol. 1419, pp. 61–91. Springer, Heidelberg (1998)
23. Rose, E.: Lightweight bytecode verification. Journal of Automated Reasoning 31, 303–334 (2003)
24. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. 12(1), 157–171 (1986)