On the Automated Synthesis of Proof-Carrying Temporal Reference Monitors

Simon Winwood[1,2], Gerwin Klein[1,2], and Manuel M. T. Chakravarty[1]

[1] University of New South Wales
School of Computer Science & Engineering
Sydney, Australia
[2] National ICT Australia[⋆]
{sjw,chak}@cse.unsw.edu.au
gerwin.klein@nicta.com.au

Abstract. We extend the range of security policies that can be guaranteed with proof carrying code from the classical type safety, control safety, memory safety, and space/time guarantees to more general security policies, such as general resource and access control. We do so by means of (1) a specification logic for security policies, which is the past-time fragment of LTL, and (2) a synthesis algorithm generating reference monitor code and accompanying proof objects from formulae of the specification logic. To evaluate the feasibility of our approach, we developed a prototype implementation producing proofs in Isabelle/HOL.

1 Introduction

Proof carrying code (PCC) [1] is inherently trustworthy, independent of its origin or previous opportunities for tampering. The guarantees provided by PCC are, however, not universal: they are relative to a *security policy* agreed upon by the code producer and consumer. It is the code producer's obligation to annotate the code with a *proof object* that establishes the code's compliance with the security policy. This proof object, consisting of steps in a *formal logic,* can be checked with a simple proof checker. Thus, the trustworthiness of the code can be established with mathematical rigour.

Existing research into the generation of proof-carrying code focuses on security policies which can be derived from properties of high-level languages and their type systems, such as type safety [2], control and memory safety [3], and space/time guarantees [4]. The contribution of this paper is to extend the approach to more general security policies, such as general resource and access control. An example of such a policy is one where "a user may perform an operation only if they have been granted a capability for that operation and that capability hasn't been revoked." Such properties are beyond the semantic guarantees of high-level languages; hence, we need (1) a formal device to express such policies and (2) a method for generating proof-carrying code for these policies.

[⋆] National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

To address Point (1), we introduce a fragment of LTL [5] lacking the usual future operators (*until* and *next*), which we call *propositional pure-past temporal logic* (P3TL), as a specification logic for security policies in Section 3. P3TL can express a wide range of security policies, while enabling the automatic synthesis of reference monitors [6] that enforce P3TL policies. This second property is key to solving Point (2). More precisely, in Section 5, we will give an algorithm to synthesise, firstly, a reference monitor checking a given P3TL policy, and secondly, a proof object demonstrating that the reference monitor code indeed meets the policy. Such a reference monitor in conjunction with a framework to constrain application code to abide by the rules of a reference monitor is sufficient to produce PCC for P3TL policies. To be complete, this framework must also provide machine checkable proof that the application code cannot subvert the reference monitor. We introduced one such framework based on hybrid sandboxing accompanied by a proof in the theorem prover Isabelle/HOL in previous work [7]. We will describe this set up in more detail in Section 2.

The proof of compliance of synthesised reference monitors is in a Hoare-like logic discussed in Section 4 and formalised in Isabelle/HOL. We implemented a prototype synthesis tool in Isabelle/HOL to demonstrate the practical feasibility of our approach.

In summary, our specific contributions are these:

– a formalisation of a simple language and program logic for reference monitors (Section 4); and,
– a synthesis algorithm for reference monitors and proof objects from P3TL formulae (Section 5).

In contrast to previous work on generating reference monitors from temporal logics, we simultaneously generate a proof object that demonstrates that the generated code enforces the security policy. We discuss related work further in Section 6.

2 Our Approach

Fig. 1 shows an application scenario in which our technique is useful. Assume a code producer generating a PCC program, certified for some policy ϕ. The code consumer, however, requires a stronger policy, ψ, where the additional guarantees of ψ over ϕ are beyond those that can be directly included during PCC generation. The method introduced in this paper enables the synthesis of a reference monitor for ψ; this can then be inserted into the PCC program, possibly by rewriting parts of the application, to bridge the gap between ϕ and ψ. This paper presents an algorithm for synthesising such monitors with matching proofs. In previous work [7], we showed how such a monitor can be integrated into an existing application in the special case where that existing application does not use PCC at all; i.e., ϕ is empty. The general case, depicted in Fig. 1, where two policies need to be integrated is left for future work.

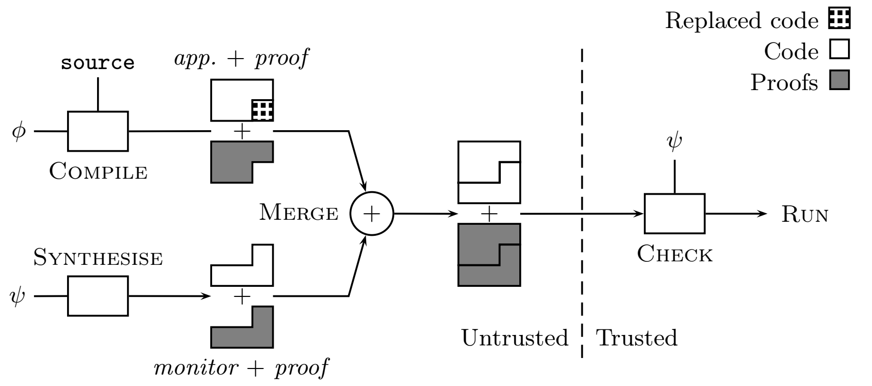

Fig. 1. A possible application of our approach: generate a monitor for the stronger policy ψ and insert it into the application. This paper presents the monitor and proof synthesis component.

2.1 The Monitor Environment

We require that the surrounding PCC infrastructure ensures the monitor invariant, containing assertions about the monitor state, be maintained outside the monitor; in a typical system, we expect the monitor state to be hidden from the rest of the system, either through a module mechanism or through general memory safety — the latter is the option we explored in [7]. If a to be secured operation occurs outside the monitor, the invariant will, in general, be invalidated — this will be detected at proof check time, and the program rejected. Of course, it is perfectly valid for there to be secured operations outside the monitor if the outside code can ensure that the invariant is maintained and that assurance is reflected in the code's proof annotations.

2.2 The Prototype

We have implemented a prototype synthesis tool. The tool generates an Isabelle theory file containing both the monitor code and proofs. The tool is Isabelle-specific, however we expect an implementation for another logical framework to have many similarities. The prototype, along with the Isabelle theory file, is freely available.[1]

While we make extensive use of Isabelle, we have been careful to ensure that there is no intrinsic reliance upon any particular feature. In the generated proofs, we avoid, where possible, use of Isabelle's automated tactics, using them only when the lemma to be proved is Isabelle-specific. This includes, for example, lemmas relating to substitution: because we make use of the meta-logic's substitution (we use a shallow embedding for the assertion logic), the proof will be Isabelle-specific, and thus no general proof is available. Conversely, the monitor's proof obligations are independent of the particular meta-logic, and so we use individual proof rules for the main part of the proof.

[1] `http://www.cse.unsw.edu.au/~sjw/papers/synthesis.html`

3 Policy Logic

In this section we introduce the logic we use for describing security policies.

Example 1. Consider a simple version of the Chinese Wall security policy[8]

> *A user may access files for any client, but once they have done so, they may only access files for that client.*

Given an operator which denotes "at some time in the past", this policy may be expressed as[2]

$$\mathtt{access}(f) \wedge f \in C \longrightarrow \neg(\mathtt{access}(g) \wedge g \notin C)$$

which states a policy equivalent to the one above: access to an object belonging to a client is allowed only if at no point in the past has the user accessed an object belonging to another client.

Example 2. Now consider a policy modelling a capability system

> *A user may perform an operation only if they have been granted a capability for that operation and that capability hasn't been revoked.*

We may encode this policy as follows

$$\mathtt{operate}_o \longrightarrow \neg\, \mathtt{revoke}_o\, \mathcal{S}\, \mathtt{grant}_o$$

where $\psi\, \mathcal{S}\, \phi$ means ϕ was true at some point in the past, and ψ ever since.

These policies, like many others, express properties over a *series* of events: the first policy that some event had not occurred in the past; the second policy that some event had occurred, and in the meantime another had not.

Given our general goal of synthesising reference monitors, our policy logic needs to fulfill two requirements: (1) we must be able to use reference monitors to implement all policies that are expressible in the logic and (2) the logic must be powerful enough to express common policies, such as those above.

The latter means that the logic must be able to express properties over behaviours of the program, not just over individual states. The former means that the logic should express safety properties only, as liveness properties cannot be implemented by reference monitors [6].

These two requirements still leave some choice. We have settled on the safety fragment [5] of propositional linear temporal logic (LTL); for the remainder of the paper we will denote this fragment *propositional pure-past temporal logic* (P3TL). As the name signifies, this is a propositional logic containing temporal operators that refer solely to previous worlds. The logic differs from traditional temporal logics in that it lacks the usual future (*until* and *next*) operators — a formula ψ in P3TL is equivalent to $\Box\psi$ in the LTL of Manna and Pnueli [5]. A similar logic (ptLTL) is used by Havelund and Rosu [9].

[2] In this section, to simplify the formalisations, we are a little loose with syntax.

$$\begin{array}{lll}
\langle\sigma\texttt{, n}\rangle \models \texttt{«a»} & = & \texttt{a}\ \sigma_{[n]} \\
\langle\sigma\texttt{, n}\rangle \models \varphi\ \texttt{[}\Rightarrow\texttt{]}\ \psi & = & \langle\sigma\texttt{, n}\rangle \models \varphi \longrightarrow \langle\sigma\texttt{, n}\rangle \models \psi \\
\langle\sigma\texttt{, n}\rangle \models \varphi\ \mathcal{S}\ \psi & = & \exists\texttt{i}\leq\texttt{n.}\ \langle\sigma\texttt{, i}\rangle \models \psi \wedge (\forall\texttt{j}\in\texttt{(i..n].}\ \langle\sigma\texttt{, j}\rangle \models \varphi) \\
\langle\sigma\texttt{, n}\rangle \models \quad \varphi & = & \texttt{n} \neq \texttt{0} \longrightarrow \langle\sigma\texttt{, n - 1}\rangle \models \varphi
\end{array}$$

Fig. 2. Semantics of P3TL

A more powerful logic, such as a first-order variant of P3TL, would enable more policies and more succinct versions of those already mentioned. Unfortunately, synthesising reference monitors for these logics is much more difficult than for P3TL — the state space for P3TL is fixed, while that of a first-order logic is potentially unbounded. We leave extension of our system to more powerful logics as future work.

The following is the syntax of P3TL

$$\texttt{form} ::= \texttt{«atom»} \mid \texttt{form}_1\ \texttt{[}\Rightarrow\texttt{]}\ \texttt{form}_2 \mid \texttt{form}_1\ \mathcal{S}\ \texttt{form}_2 \mid \quad \texttt{form}$$

The logic consists of atoms and the operators implication, since, and weak previous. Implication is the usual binary operator; to distinguish it from that of HOL, we write $\varphi\ \texttt{[}\Rightarrow\texttt{]}\ \psi$. The syntax for the since and weak previous operators is standard.

The definition of P3TL is parameterised over the type of states; atoms are HOL predicates on such states. Our implementation of P3TL is then a *deep embedding* into Isabelle/HOL, but uses a *shallow embedding* for atomic propositions.

Example 3. A traditional state space would be a tuple or record of variables. The predicate `«λs. x s = 7»` then, for instance, states that the variable `x` in record `s` should have value `7`.

The following gives the definitions for the other propositional connectives (negation, conjunction, disjunction, etc.) as well as the usual temporal operators: *strong previous* (φ), *once* (φ), and *so-far* (φ). The two previous operators (strong and weak) differ only at the initial state: φ is false and φ is true. In particular, $\bot$ is true only at the initial state.

$$\begin{array}{lllllll}
\top & = & \texttt{«}\lambda\texttt{s. True»} & \quad \varphi & = & \texttt{[}\neg\texttt{] (([}\neg\texttt{]}\ \varphi\texttt{))} & \text{(Previously)} \\
\bot & = & \texttt{«}\lambda\texttt{s. False»} & \quad \varphi & = & \top\ \mathcal{S}\ \varphi & \text{(Once)} \\
\texttt{[}\neg\texttt{]}\ \varphi & = & \varphi\ \texttt{[}\Rightarrow\texttt{]}\ \bot & \quad \varphi & = & \texttt{[}\neg\texttt{] (([}\neg\texttt{]}\ \varphi\texttt{))} & \text{(So-far)} \\
\varphi\ \texttt{[}\wedge\texttt{]}\ \psi & = & \texttt{[}\neg\texttt{] (}\varphi\ \texttt{[}\Rightarrow\texttt{] [}\neg\texttt{]}\ \psi\texttt{)} & \varphi\ \texttt{[}\vee\texttt{]}\ \psi & = & \texttt{[}\neg\texttt{]}\ \varphi\ \texttt{[}\Rightarrow\texttt{]}\ \psi &
\end{array}$$

Fig. 2 gives the semantics of P3TL using an *indexed model* $\langle\sigma\texttt{,n}\rangle$. The first element, σ, is a sequence of worlds, and the second, `n`, is the index of the current state.

An atom `«a»` is valid in $\langle\sigma\texttt{,n}\rangle$ iff the function `a` maps the n-th state in σ (denoted $\sigma_{[n]}$) to true. An implication is valid iff the validity of the premise implies the validity of the conclusion at the same index `n`. The formula $\varphi\ \mathcal{S}\ \psi$ is valid at $\langle\sigma\texttt{,n}\rangle$ iff ψ was valid at some earlier state `i` and φ was valid at all states `j` from `i` to `n`. The weak previous operator φ is valid iff there is either no previous state (`n = 0`) or φ was valid at index `n - 1`.

4 A Language and Logic for Reference Monitors

4.1 The Programming Language

This section describes the language we use for reference monitors. It is a simple imperative if-while language, using Isabelle/HOL's functions for expressions. Whilst it is usual in expositions on proof-carrying code systems to use a very low-level language (e.g., an assembly language), we chose this comparatively high-level language to simplify the presentation; issues such as memory allocation are orthogonal to the contribution of this work and would only obscure the central points. We expect that low-level languages will present few additional theoretical hurdles — indeed, Barthe et. al. [10] note that, in the absence of optimisations, transformation to a low-level language preserves proof obligations.

The syntax of programs is shown below. The nonterminal *basic* denotes functions from states to states, *bexp* functions from states to booleans, and *form* P3TL formulae.

$$
\begin{aligned}
\mathit{stmt} ::= {} & \texttt{Do}\ \mathit{basic} \quad | \quad \mathit{stmt}_1\ ;\ \mathit{stmt}_2 \\
| \ & \texttt{IF}\ \mathit{bexp}\ \texttt{THEN}\ \mathit{stmt}_1\ \texttt{ELSE}\ \mathit{stmt}_2\ \texttt{FI} \quad | \quad \texttt{WHILE}\ \mathit{bexp}\ \texttt{DO}\ \mathit{stmt}\ \texttt{OD} \\
| \ & \texttt{Secure}\ \mathit{form}
\end{aligned}
$$

As with P3TL, the syntax is parameterised over the states of the program. The `Do` statement provides the means to model assignment and simple state transformations directly as HOL functions. The `Secure` statement represents the operation to be secured, abstracting away from the particular operation's semantics. It is parameterised by a P3TL formula representing the security policy.

Example 4. The following monitor checks the Chinese Wall policy from Example 1

```
IF (λs. ∃f. access s f ∧ f ∈ C) THEN
  IF (λs. seen s = 1) THEN
    Do (λs. s(|error := 1|))
  ELSE
    Secure ψ
  FI
ELSE
  Secure ψ
FI
```

The state variable `seen` records whether we have previously seen a conflicting access — the code to maintain this is omitted. The expression `s(|error := 1|)` updates `error` in record `s`.

The states over which the language operates are tuples $(\texttt{s}, \sigma)$, where `s` is the program state, usually modelled as an Isabelle record[3], and σ a sequence

[3] As we use the Do statement to model assignment, taking a state update function as an argument, there is no requirement that a record is used — it merely simplifies use of the language.

$$\boxed{\mathtt{s} \xrightarrow{\mathtt{t}} \mathtt{s'}}$$

$$\frac{}{\mathtt{(s,\ ss)} \xrightarrow{\mathtt{Do\ f}} \mathtt{(f\ s,\ ss)}} \qquad \frac{\mathtt{s} \xrightarrow{\mathtt{e}} \mathtt{t} \quad \mathtt{t} \xrightarrow{\mathtt{e'}} \mathtt{s'}}{\mathtt{s} \xrightarrow{\mathtt{e;\ e'}} \mathtt{s'}}$$

$$\frac{\mathtt{b\ (fst\ s)} \quad \mathtt{s} \xrightarrow{\mathtt{e}} \mathtt{s'}}{\mathtt{s} \xrightarrow{\mathtt{IF\ b\ THEN\ e\ ELSE\ e'\ FI}} \mathtt{s'}} \qquad \frac{\neg\ \mathtt{b\ (fst\ s)} \quad \mathtt{s} \xrightarrow{\mathtt{e'}} \mathtt{s'}}{\mathtt{s} \xrightarrow{\mathtt{IF\ b\ THEN\ e\ ELSE\ e'\ FI}} \mathtt{s'}}$$

$$\frac{\neg\ \mathtt{b\ (fst\ s)}}{\mathtt{s} \xrightarrow{\mathtt{WHILE\ b\ DO\ e\ OD}} \mathtt{s}}$$

$$\frac{\mathtt{b\ (fst\ s)} \quad \mathtt{s} \xrightarrow{\mathtt{e}} \mathtt{t} \quad \mathtt{t} \xrightarrow{\mathtt{WHILE\ b\ DO\ e\ OD}} \mathtt{s'}}{\mathtt{s} \xrightarrow{\mathtt{WHILE\ b\ DO\ e\ OD}} \mathtt{s'}}$$

$$\frac{\mathtt{s'} = (\mathsf{if}\ \mathtt{(s,\ ss)} \models \varphi\ \mathsf{then}\ \mathtt{(s,\ ss\ @\ [s])}\ \mathsf{else}\ \mathtt{arbitrary})}{\mathtt{(s,\ ss)} \xrightarrow{\mathtt{Secure}\ \varphi} \mathtt{s'}}$$

Fig. 3. Semantics of programs

representing a trace of previously seen states. This trace serves as the model when checking P3TL formulae; it records the state at security events only, not all state changes in the program. Note that the trace is only required for reasoning about policies and does *not* appear at runtime.

In monitoring P3TL policies, we interpret the validity of formulae relative to the current state, that is, the last state in the sequence. We thus use an *anchored interpretation* [11]. Satisfaction of P3TL formulae by a program state is then

$$\mathtt{(s,}\ \sigma\mathtt{)} \models \varphi = \langle \sigma\ \mathtt{@\ [s],}\ |\sigma| \rangle \models \varphi$$

Fig. 3 shows a big step semantics for our language. The semantics are standard [12], apart from the `Secure` statement: the effect of `Secure` φ is to record the current state in the state history. Execution of `Secure` φ is only defined, however, when the current state satisfies the security policy φ.

A full implementation of our system would replace the `Secure` statement by, for example, a system call statement. Again, modelling the behaviour of system calls is orthogonal to the aims of this paper; the much-simplified `Secure` is sufficient.

4.2 The Program Logic

So far we have defined a logic for policies and a language to implement reference monitors. This section introduces a logic for reasoning reasons about programs that allows us to formally verify that safety policies are respected.

The rules of this Hoare-like program logic are shown in Fig 4. The triple $\Vdash$ `{P} T {Q}` denotes that a statement `T` that starts execution in a state satisfying `P` and terminates will finish in a state satisfying `Q`. Both `P` and `Q` are assertions, that is, HOL predicates on states.

As with the semantics in Sect. 4.1, the proof rules are standard apart from the `Secure` statement. We have one rule for each syntactic construct and the usual rule of consequence.

$$\boxed{\Vdash \{P\}\ T\ \{Q\}}$$

$$\frac{}{\Vdash \{\lambda(s,\ ss).\ P\ (f\ s,\ ss)\}\ \mathtt{Do}\ f\ \{P\}}\ \text{DO} \qquad \frac{\Vdash \{P\}\ e\ \{Q\} \qquad \Vdash \{Q\}\ e'\ \{R\}}{\Vdash \{P\}\ e;\ e'\ \{R\}}\ \text{SEQ}$$

$$\frac{\Vdash \{\lambda s.\ P\ s \wedge b\ (\mathtt{fst}\ s)\}\ e\ \{Q\} \qquad \Vdash \{\lambda s.\ P\ s \wedge \neg\ b\ (\mathtt{fst}\ s)\}\ e'\ \{Q\}}{\Vdash \{P\}\ \mathtt{IF}\ b\ \mathtt{THEN}\ e\ \mathtt{ELSE}\ e'\ \mathtt{FI}\ \{Q\}}\ \text{COND}$$

$$\frac{\Vdash \{\lambda s.\ P\ s \wedge b\ (\mathtt{fst}\ s)\}\ e\ \{P\} \qquad Q = (\lambda s.\ P\ s \wedge \neg\ b\ (\mathtt{fst}\ s))}{\Vdash \{P\}\ \mathtt{WHILE}\ b\ \mathtt{DO}\ e\ \mathtt{OD}\ \{Q\}}\ \text{WHILE}$$

$$\frac{\forall s.\ P'\ s \longrightarrow P\ s \qquad \forall s.\ Q\ s \longrightarrow Q'\ s \qquad \Vdash \{P\}\ e\ \{Q\}}{\Vdash \{P'\}\ e\ \{Q'\}}\ \text{CONS}$$

$$\frac{\forall s.\ P\ s \longrightarrow s \models \varphi \qquad \forall s\ ss.\ P\ (s,\ ss) \longrightarrow Q\ (s,\ ss\ @\ [s])}{\Vdash \{P\}\ \mathtt{Secure}\ \varphi\ \{Q\}}\ \text{SECURE}$$

Fig. 4. A Hoare-like logic for the programming language

The new rule for `Secure` demands that all states that satisfy the precondition be models of the security policy φ; this check links the assertion logic to P3TL, the policy logic. In addition, the rule's postcondition reflects the effect of `Secure` on the program state, i.e. recording the event in the trace.

Following the standard practise [12] of using a shallow embedding of assertions into Isabelle/HOL means that we can take direct advantage of Isabelle's tactics and libraries to reason about programs. The assertion logic and program expressions, however, are more powerful than strictly required[4].

We have shown this Hoare-logic to be sound in the following sense.

Theorem 1. If $\Vdash \{P\}\ e\ \{Q\}$ and $P\ s$ and $s\ {-e\!\!\rightarrow}\ s'$ then $Q\ s'$.

Note that our definition of the semantics of `Secure` ensures that programs cannot get stuck because of policy violation. The only reason that there might be no `s'` such that $s\ {-e\!\!\rightarrow}\ s'$ in this theorem is non-termination of while loops.

5 Synthesis

In this section we discuss the synthesis algorithm; that is, the algorithm that takes a P3TL formula and emits a reference monitor which enforces the policy, and a proof that the generated monitor does indeed enforce the policy.

We will use the example policy

$$(\langle\!\langle \lambda s.\ x\ s = 1 \rangle\!\rangle)\ [\Rightarrow]\ (\langle\!\langle \lambda s.\ x\ s < 5 \rangle\!\rangle\ \mathcal{S}\ \langle\!\langle \lambda s.\ y\ s = 1 \rangle\!\rangle) \qquad (*)$$

[4] We also have a deep embedding for all logics. We use the shallow embedding as it is much briefer and clearer: the interaction between the assertion logic and P3TL, although non-trivial, is not the main focus of this paper.

as a running example throughout this section. Although this example has no correspondence to a real-world policy (even a small such policy would be too big for the limited space available), it contains enough complexity to be of interest.

5.1 Checking P3TL Satisfaction

In the synthesis of P3TL formulae, the following definition is required

Definition 1. *The* past formulae *of a formula are sub-formulae*

1. *of the form $\phi \mathcal{S} \psi$; or*
2. *that occur after , e.g. ψ in ψ.*

The number of sub-formulae is linear in the size of the formula.

Example 5. The past formulae for our example policy (*) are `«λs. x s = 1»` and `«λs. x s < 5»` $\mathcal{S}$ `«λs. y s = 1»`. Terms in our examples which relate to these formulae will have suffixes 0 and 1, respectively. For example, `state_0` is $\Sigma_{\text{«}\lambda\texttt{s. x s = 1}\text{»}}$.

Note that a since formula may be unfolded according to the equality:

$$\phi \mathcal{S} \psi = \psi\, [\vee] ((\phi \mathcal{S} \psi)\, [\wedge]\, \phi)$$

Applying this rule repeatedly yields a formula in which all since formulae occur only after a operator. Thus, truth of the rewritten formula depends only on the truth of propositions in the current world and the truth of sub-formulae in the immediately previous world. Furthermore, note that these sub-formulae are all past formulae.

To check if a sequence σ satisfies a policy, we start by setting all formulae of the form φ to false (as per the semantics of). Starting from the initial state, check and record the truth of all past formulae. At the last state in σ, check the truth of the rewritten policy, using the recorded past formulae from the previous state.

Thus, to monitor a policy, we need only keep track of its past formulae. This implies that we do not require an explicit representation of worlds in our monitor, and that checking a P3TL formula can be done efficiently.

5.2 Monitor Synthesis

The algorithm for constructing a monitor for a P3TL formula is then as follows:

1. For each past formulae ψ, allocate a state bit, Σ_ψ, which records the truth of the formula in the previous world, i.e. $\Sigma_\psi \leftrightarrow \psi$;
2. Construct a program fragment which checks the truth of each past formula ψ in the current world, i.e., with respect to the current program state. When constructing the fragment, if a sub-formula of ψ of the form ϕ is seen, emit code which checks the state bit Σ_ϕ;

```
IF (λs. x s = 1) THEN
  Do (λs. s⦇tmp_0 := 1⦈)
ELSE
  Do (λs. s⦇tmp_0 := 0⦈)
FI
IF (λs. y s = 1) THEN
  Do (λs. s⦇tmp_1 := 1⦈)
ELSE
  IF (λs. state_1 s = 1) THEN
    IF (λs. x s < 5) THEN
      Do (λs. s⦇tmp_1 := 1⦈)
    ELSE
      Do (λs. s⦇tmp_1 := 0⦈)
    FI
  ELSE
    Do (λs. s⦇tmp_1 := 0⦈)
  FI
FI
```

(CONT.)

```
IF (λs. state_0 s = 1) THEN
  IF (λs. tmp_1 s = 1) THEN
    Secure policy;
    Do (λs. s⦇state_0 := tmp_0 s⦈);
    Do (λs. s⦇state_1 := tmp_1 s⦈)
  ELSE
    Skip
  FI
ELSE
  Secure policy;
  Do (λs. s⦇state_0 := tmp_0 s⦈);
  Do (λs. s⦇state_1 := tmp_1 s⦈)
FI
```

Fig. 5. Generated code for policy (*). The left hand column contains the state maintenance code, while the right side contains the policy checking code.

3. Construct a monitor fragment for the main formula. In the case that the formula holds, execute the secure statement and update the monitor state, otherwise handle the security violation; and
4. Sequentially compose the fragments to generate the final monitor.

Fig. 6 presents the algorithm for constructing a monitor fragment. The notation $\mathcal{S}[\![\psi]\!]_{(tc,fc)}$ denotes the algorithm applied to formula ψ, with arguments `tc` (true case) and `fc` (false case). The leaves of the fragment, `tc` and `fc`, are assignments in the case of a past formula, and a secure statement along with state update in the case of the policy.

Example 6. The monitor generated for our example policy is shown in Fig. 5. Our implementation includes an optimisation: rather than re-check a formula, monitor fragments may refer to previously established past formula by checking the corresponding variable.

The monitor fragment for the past formula `«λs. x s = 1»` (the first 5 lines of Fig. 5) was generated by

$$\mathcal{S}[\![\texttt{«}\lambda\texttt{s. x s = 1»}]\!]_{(\texttt{Do (}\lambda\texttt{s. s⦇tmp\_0 := 1⦈)},\texttt{Do (}\lambda\texttt{s. s⦇tmp\_0 := 0⦈)})}$$

Not shown are the proof handling aspects; in our prototype, the `Do` statements are functions taking a proof of `«λs. x s = 1»` and `¬«λs. x s = 1»` respectively. These proofs are then stored for later extraction.

If we, for the moment, ignore proof generation, the synthesis algorithm is straightforward: when an atom is seen, generate code which checks the atom and uses

$$
\begin{aligned}
&\mathcal{S}[\![«a»]\!]_{(tc,\,fc)} &&= \texttt{IF}\ a\ \texttt{THEN}\ (tc\ \textsc{atom}\mathrm{I})\ \texttt{ELSE}\ (fc\ \textsc{natom}\mathrm{I})\\
&\mathcal{S}[\![\psi\ [\Rightarrow]\ \phi]\!]_{(tc,\,fc)} &&= \textbf{let}\\
& &&\qquad tc' \ = \ \lambda\, r.\mathcal{S}[\![\phi]\!]_{(tc\,\cdot\,\textsc{impl}\mathrm{I}_1,\ fc\,\cdot\,(\textsc{nimpl}\mathrm{I}\ r))}\\
& &&\quad \textbf{in}\\
& &&\quad \mathcal{S}[\![\psi]\!]_{(tc',\ tc\,\cdot\,\textsc{impl}\mathrm{I}_2)}\\
&\mathcal{S}[\![[\neg]\psi]\!]_{(tc,\,fc)} &&= \mathcal{S}[\![\psi]\!]_{(fc\,\cdot\,\textsc{nneg}\mathrm{I},\ tc\,\cdot\,\textsc{neg}\mathrm{I})}\\
&\mathcal{S}[\![\psi]\!]_{(tc,\,fc)} &&= \texttt{IF}\ (\Sigma_\psi)\ \texttt{THEN}\ (tc\ \textsc{inv}\text{-}\pi_\psi)\ \texttt{ELSE}\ (fc\ \textsc{ninv}\text{-}\pi_\psi)\\
&\mathcal{S}[\![\psi\ \mathcal{S}\ \phi]\!]_{(tc,\,fc)} &&= \mathcal{S}[\![\phi\ [\vee]((\psi\ \mathcal{S}\ \phi)\ [\wedge]\ \psi)]\!]_{(tc\,\cdot\,\textsc{since}\mathrm{I},\ fc\,\cdot\,\textsc{nsince}\mathrm{I})}
\end{aligned}
$$

Fig. 6. The algorithm for constructing a monitor fragment. It generates both a monitor fragment and proof annotations. The parameters `tc` and `fc` are functions which take proof annotations and return the statements to be used in each branch of the conditional; the term `tc` · SINCEI composes rule SINCEI with `tc`, thus adding a new rule to the proof tree at `tc`.

$$\boxed{\mathtt{s} \models \varphi}$$

$$
\frac{\varphi\ \texttt{(fst s)}}{\mathtt{s} \models «\varphi»}\,\textsc{atom}\mathrm{I}
\qquad
\frac{\neg\ \varphi\ \texttt{(fst s)}}{\neg\ \mathtt{s} \models «\varphi»}\,\textsc{natom}\mathrm{I}
$$

$$
\frac{\mathtt{s} \models \varphi}{\mathtt{s} \models \psi\ [\Rightarrow]\ \varphi}\,\textsc{impl}\mathrm{I}_1
\qquad
\frac{\neg\ \mathtt{s} \models \psi}{\mathtt{s} \models \psi\ [\Rightarrow]\ \varphi}\,\textsc{impl}\mathrm{I}_2
\qquad
\frac{\mathtt{s} \models \varphi \qquad \neg\ \mathtt{s} \models \psi}{\neg\ \mathtt{s} \models \varphi\ [\Rightarrow]\ \psi}\,\textsc{nimpl}\mathrm{I}
$$

$$
\frac{\neg\ \mathtt{s} \models \varphi}{\mathtt{s} \models [\neg]\ \varphi}\,\textsc{neg}\mathrm{I}
\qquad
\frac{\mathtt{s} \models \varphi}{\neg\ \mathtt{s} \models [\neg]\ \varphi}\,\textsc{nneg}\mathrm{I}
$$

$$
\frac{\mathtt{s} \models \psi\ [\vee]\ (\ (\varphi\ \mathcal{S}\ \psi)\ [\wedge]\ \varphi)}{\mathtt{s} \models \varphi\ \mathcal{S}\ \psi}\,\textsc{since}\mathrm{I}
$$

$$
\frac{\neg\ \mathtt{s} \models \psi\ [\vee]\ (\ (\varphi\ \mathcal{S}\ \psi)\ [\wedge]\ \varphi)}{\neg\ \mathtt{s} \models \varphi\ \mathcal{S}\ \psi}\,\textsc{nsince}\mathrm{I}
$$

Fig. 7. Derived introduction rules for P3TL

`tc` in the true case, `fc` in the false case; in the case of a negation, $[\neg]\psi$, generate code for ψ but switch the leaves — if ψ holds, then use `fc`, otherwise `tc`. Note that negation produces no extra code; it merely swaps leaves. This means that special cases for conjunction and disjunction are not required: unfolding the definitions results in no additional code; a previous formula, ψ, results in a check of the state component for that formula, Σ_ψ.

5.3 Proof Synthesis

The main monitor theorem is

$$\Vdash\ \{\textsc{inv}_\psi\}\ \texttt{monitor}\ \{\textsc{inv}_\psi\}$$

which states that the monitor preserves the invariant, discussed below. The proof that the monitor enforces the security policy is implicit in the use of any `Secure` statements: wherever an operation occurs in the monitor, a proof obligation is required which states that the policy holds (c.f. rule SECURE). Thus, as our proof system is sound (Theorem 1), a proof of the main theorem implies that execution of a `Secure` statement occurs only when the security policy is true.

The proof of the main theorem is generated from the monitor fragments using a verification condition generator style algorithm. The proof requires a number of lemmas for both re-establishing the monitor invariant and for showing the policy holds for `Secure` statements. The generation of these proof obligations is discussed below.

The monitor invariant. The monitor maintains state between invocations, in particular Σ_ϕ for each past formula ϕ, and so a monitor invariant is required. This invariant relates the value of each state variable to the truth of the corresponding formula in the previous state.

$$\text{INV}_\psi \equiv \bigwedge_{\phi \in past(\psi)} \Sigma_\phi \leftrightarrow \phi$$

where ψ is the security policy and $past(\psi)$ are the past formulae of ψ.

The past formula monitor fragments are then responsible for reestablishing the monitor invariant. This is done in two steps: firstly, the monitor checks the past formula, ϕ, and sets a temporary variable, Δ_ϕ, accordingly; secondly, the monitor updates the real monitor state *after* execution of the `Secure` statement.

This two-step process is required for a number of reasons: primarily, later monitor fragments may require the state variable to establish the truth of other formulae, past or policy; and, secondly, if the security policy doesn't hold, then the monitor may elect to silently ignore the request. In this case, the initial value of the state variables is still correct — no secure operation is performed, and thus the invariant is still true. This is the behaviour of monitors generated by our prototype. We then define, for each past formula, a pre-invariant $\Xi_\phi \equiv \Delta_\phi \leftrightarrow \phi$. This correspondence is used to generate the invariant — after execution of `Secure` statement and state update, we can use this to derive $\Sigma_\phi \leftrightarrow \phi$

Example 7. Our prototype generates a number of auxiliary lemmas for manipulating the invariant. These include

– projection rules (the INV-π_ϕ rules mentioned in Fig. 6)

$$\frac{\texttt{invariant (s, }\sigma\texttt{)} \qquad \texttt{state\_1 s = 1}}{\texttt{(s, }\sigma\texttt{)} \models \texttt{ («}\lambda\texttt{s. x s < 5»}\ \mathcal{S}\ \texttt{«}\lambda\texttt{s. y s = 1»)}}\ \text{INV-}\pi_1$$

– rules for invariance under assignment (`X` is an arbitrary function)

$$\texttt{invariant (s(|tmp\_1 := X s|), }\sigma\texttt{) = invariant (s, }\sigma\texttt{)}$$

Proof obligations. The proof of the main theorem requires, for each assignment in each past formula monitor fragment, a proof obligation of the form

$$\text{INV}_{\psi} \wedge \Xi_{\phi_1} \wedge \ldots \wedge \Xi_{\phi_m} \wedge a_1 \wedge \ldots \wedge a_n \longrightarrow \text{INV}'_{\psi} \wedge \Xi'_{\phi_1} \wedge \ldots \wedge \Xi'_{\phi_m} \wedge \Xi'_{\phi}$$

A formula is primed to denote it's truth after the assignment. The terms $\Xi_{\phi_1} \wedge \ldots \wedge \Xi_{\phi_m}$ are the past formula equivalences established by previous monitor fragments, $a_1, \ldots, a_n$ are the atoms (or their negation, in the case of a false branch) that were checked by the conditionals in the current monitor fragment, and ϕ is the formula from which the monitor fragment was produced, with Δ_{ϕ} replaced by `True` or `False` depending on which branch the assignment occurs.

These proofs state that the monitor fragment correctly establishes Ξ_{ϕ} and does not invalidate previously established equivalences or the invariant.

The obligations for `Secure` statements are similar but for the last term in the conjunction: in this case, it is the policy. In addition, the SECURE rule allows us to generate P3TL terms of the form ϕ, assuming we have ϕ — this is how the conversion from pre-invariant to invariant occurs.

Example 8. The following obligation is generated for the fourth assignment in Fig. 5.

```
∀s σ. invariant (s, σ) ∧
      pre_sd0 (s, σ) ∧ y s ≠ 1 ∧ state_1 s = 1 ∧ x s < 5 ⟶
      invariant (s(|tmp_1 := 1|), σ) ∧
      pre_sd0 (s(|tmp_1 := 1|), σ) ∧ pre_sd1 (s(|tmp_1 := 1|), σ)
```

Proof construction. Much of complexity of the algorithm in Fig. 6 arises because the leaves of the tree are annotated with the proof, and the algorithm builds the tree from the leaves up. The proofs require information that is not initially available, i.e., the proofs for sub-formulae, so the arguments to the synthesis function, `tc` and `fc`, are *functions* from proofs to program fragments. In particular, if the fragment is checking the truth of ϕ, `tc` will be a function from a proof of ϕ, and `fc` a function from a proof of $\neg\phi$.

Fig. 7 shows the P3TL proof rules used by the algorithm, except for the invariant projection lemmas which are described above. All of these rules are derived from the semantics of P3TL (using Isabelle) as lemmas, but are intended to be derivable in a syntactic proof system (such as that in Lichtenstein and Pnueli [13]). Each rule occurs in the positive and negated form, as some proof rules require negated forms of past formulae (e.g., rule NINV-π_{ϕ}). Note that the atom rules convert assertions into P3TL atoms.

The algorithm assumes these rules are available as functions which build proofs. As the assumptions of the proof are assertions (the invariant and $a_1, \ldots, a_n$ as above), those rules without premises of the form `s` $\models \psi$ are treated as constants, as in the case for atoms.

In the case of an implication, $\psi\ [\Rightarrow]\ \phi$, we construct a function tc' which takes as argument a proof of ψ and produces a fragment which checks ϕ. Note that NIMPI is partially applied to the proof of ψ in the false case; the result is a function from $\neg\phi$ as required. The remaining cases are straightforward.

Example 9. The following is the proof generated for the obligation in Example 8

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{invariant (s, }\sigma\texttt{)}\quad \texttt{state\_1 s = 1}}{\texttt{(s, }\sigma\texttt{)} \models \ominus\ \varphi}\,\textsc{inv-}\pi_1 \qquad \dfrac{\dfrac{\texttt{y s = 1}}{\texttt{(s, }\sigma\texttt{)} \models \text{«}\lambda\texttt{s. y s = 1}\text{»}}\,\textsc{atomI}}{\neg\ \texttt{(s, }\sigma\texttt{)} \models \texttt{[}\neg\texttt{]}\ \text{«}\lambda\texttt{s. y s = 1}\text{»}}\,\textsc{nnegI}}{\neg\ \texttt{(s, }\sigma\texttt{)} \models \psi}\,\textsc{nimplI}}{\texttt{(s, }\sigma\texttt{)} \models \texttt{[}\neg\texttt{]}\ \psi}\,\textsc{negI}}{\texttt{(s, }\sigma\texttt{)} \models \texttt{[}\neg\texttt{]}\ \text{«}\lambda\texttt{s. y s = 1}\text{»}\ \texttt{[}\Rightarrow\texttt{]}\ \texttt{[}\neg\texttt{]}\ \psi}\,\textsc{implI1}}{\texttt{(s, }\sigma\texttt{)} \models \varphi}\,\textsc{sinceI}}{\texttt{(tmp\_1 (s(|tmp\_1 := 1|)) = 1) = (s, }\sigma\texttt{)} \models \varphi}\,\textsc{tmp-1-supdT}
$$

where ψ = («λs. x s < 5» $\mathcal{S}$ «λs. y s = 1») [⇒] [¬] «λs. y s = 1» and φ = «λs. x s < 5» $\mathcal{S}$ «λs. y s = 1».

The above proof is generated in two steps: firstly (not shown) the previously established facts are shown to be true after the assignment using the assignment invariance rules described above; and secondly, shown above, the new equivalence is established using the proof constructed by the algorithm in Fig. 6.

After the assignment (tmp_1 s = 1) = (s, σ) $\models$ φ, as required.

5.4 Discussion

Tableau construction [14] algorithms give an exponential state space due to the use of subset construction, and thus can generate monitors whose worst-case size is exponential in the size of the input formula. Our approach, while tableau-based, has a worst-case size that is ($O(n^2)$) in the size of the input formula. This size reduction is due to the past formula monitor fragments which dynamically calculate the automata transitions. This comes at a cost, however: the time complexity of our approach is quadratic, while that of a simpler automata-based solution is linear in the size of the formula.

6 Related Work

Bernard and Lee [15] present a proof carrying code framework based on temporal logic. In contrast, our system is closer to that of a traditional PCC framework — we require temporal terms only when dealing with the high-level safety policy; this should make extending existing programs simpler. Nevertheless, we envisage no major issues in synthesising monitors for their framework.

Synthesis from temporal logics, traditionally used in the model checking community (e.g. Spin [16]) has gained recent popularity for constructing program reference monitors [17,18,19].

In particular, our approach is similar to that of the PathExplorer project [9]. They construct monitors for safety properties using an algorithm that is very close to that we presented. These monitors can then be automatically inserted into Java programs for run-time testing. The major difference is that our algorithm also generates proofs.

Peled and Zuck [20] generate a proof from model-checking results. This proof shows properties of the target system; in theory, we may be able to generate the monitor and then apply their technique to generate the proof. It is, however, unclear whether that approach would be feasible in practice.

7 Conclusions and Future Work

In this paper we introduced a temporal logic for formulating security policies — propositional pure past temporal logic — and showed how to automatically generate efficient reference monitor implementations that check the required policy, along with a machine-checkable proof of their safety.

We have implemented a prototype targeting Isabelle/HOL; the majority of formal matter, and all of the theorems, in this paper were generated using Isabelle's presentation mechanism [21] from the Isabelle proofs. This means what we show is what we proved.

The main contribution of this paper is to show how reference monitor synthesis, proof generation, and the policy logic are defined and interact. In future, we are interested in refining the system towards one in which the reference monitors are implemented in a low-level language and inserted into consumer code by binary rewriting, as demonstrated in [7]. Also desirable is a richer policy logic, such as a first order variant of P3TL; finally, a higher-level language that can be compiled into P3TL would ease the job of writing security policies.

Acknowledgements

We wish to thanks Kai Engelhardt and Harvey Tuch, along with the anonymous reviewers, for their valuable comments on earlier versions of this paper.

References

1. Necula, G.C.: Proof-carrying code. In: Proc. of POPL'97, Paris (1997) 106–119
2. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. TOPLAS **21** (1999) 527–568
3. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. In: Proc. of PLDI'98. Volume 33,5., New York, ACM Press (1998) 333–344
4. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., Stark, I.: Mobile resource guarantees for smart devices. In: Proc. of CASSIS'04. Volume 3362 of LNCS., Springer (2005) 1–26
5. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer-Verlag, New York (1995)
6. Schneider, F.B.: Enforceable security policies. Information and System Security **3** (2000) 30–50
7. Winwood, S., Chakravarty, M.M.T.: Secure untrusted binaries - provably!. In: Proc. of FAST'05. Volume 3866 of LNCS., Springer (2006) 171–186
8. Brewer, D.F.C., Nash, M.J.: The Chinese Wall security policy. In: IEEE Symposium on Security and Privacy. (1989) 206–214

9. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Proc. of TACAS'02. Volume 2280 of LNCS., Springer (2002) 342–356
10. Barthe, G., Rezk, T., Saabas, A.: Proof obligations preserving compilation. In: Proc. of FAST'05. Volume 3866 of LNCS., Springer (2006) 112–126
11. Manna, Z., Pnueli, A.: The anchored version of the temporal framework. In de Bakker, J.W., de Roever, W.P., Rozenberg, G., eds.: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. Volume 354 of LNCS., Springer (1989) 201–284
12. Nipkow, T.: Hoare logics in Isabelle/HOL. In Schwichtenberg, H., Steinbrüggen, R., eds.: Proof and System-Reliability, Kluwer (2002) 341–367
13. Lichtenstein, O., Pnueli, A.: Propositional temporal logics: Decidability and completeness. Logic Journal of the IGPL **8** (2000) 55–85
14. Geilen, M.: On the construction of monitors for temporal logic properties. Volume 55. (2001)
15. Bernard, A., Lee, P.: Temporal logic for proof-carrying code. In: Proc. of CADE'02, London, UK, Springer-Verlag (2002) 31–46
16. Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. **23** (1997) 279–295
17. Chen, F., d'Amorim, M., Rosu, G.: A formal monitoring-based framework for software development and analysis. In: Proc. of ICFEM'04. Volume 3308 of LNCS., Springer (2004) 357–372
18. d'Amorim, M., Rosu, G.: Efficient monitoring of omega-languages. Volume 3576 of LNCS. (2005) 364–378
19. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Program monitoring with LTL in EAGLE. In: Proc. of PADTAD'04. (2004)
20. Peled, D., Zuck, L.: From model checking to a temporal proof. In: Proc. of SPIN'01, New York, NY, USA, Springer (2001) 1–14
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)