Security Check: A Formal Yet Practical Framework For Secure Software Architecture

Arnab Ray
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
{arnabray}@cs.sunysb.edu

ABSTRACT

With security becoming an important concern for both users as well as designers of large-scale software systems, it is necessary to introduce security considerations very early in the system development life-cycle namely in the modeling phase itself. But the main problem in the widespread adoption of security modeling has been that representations of even moderate-size systems consume so much memory (due to the infamous state space explosion problem) that designers are loathe to spend time increasing the complexity of their models by introducing security aspects in the design phase itself. In this paper we propose a technique called *security check* which entails taking small units of a system, putting them in a "security harness" that exercises relevant executions appropriately within the unit, and then model checking these more tractable units. For most systems whose security requirements are localized to individual system components or interactions between small numbers of components, *security check* offers a means of coping with state explosion. Another major benefit of *security check* is that it enables us to detect system vulnerabilities even when the attack behavior is not known. And for known attack patterns *security check* can provide models of suspicious behavior which can then be used for intrusion detection at a later stage.

1. INTRODUCTION

Traditional research on security focuses on post-implementation analysis of software systems. In these approaches, source code may be annotated to find bugs [9] or be used to generate abstract models for subsequent analysis [8] [2] [12] [23]. But these approaches address the issue of security *after* the entire system has been coded and even perhaps deployed. Finding security vulnerabilities in the software design at this stage leaves us with two options: either patch the vulnerability by writing new code or go back to the drawing board and redesign the vulnerable parts of the system. While the first approach is the most common, it can be argued that this solution is not good software engineering practice as patches are at best an ad-hoc solution and a patched system may become so specialized that it makes component reuse and consequent software maintenance difficult. The second approach however leads to an increase in development time and cost which is why the first solution, despite its disadvantages, is the most popular.

This motivates our approach which concentrates only on pre-implementation models of software systems. The utility of design-time artifacts is that they allow users to model the system at any level of detail, without worrying about implementation details, in order to study high-level structure and behavior and isolate bugs early in the development life-cycle. The modeling notations typically reduce, semantically, to different variants of finite-state machines. Requirements are often given either in temporal logic [15, 19] or also as state machines. The term *model checking* [3] is often used to encompass algorithmic techniques for determining whether or not (formal) system models satisfy (formal) system requirements.

One of the principal problems with model-checking is that for model-checking sufficiently complex real-world systems, the logical representations of the systems constructed for the purpose of analysis become so large that even powerful workstations cannot handle them. The problem is compounded when the system modeled has real-time characteristics. The added obligation of tracking delays requires the introduction of states. This compounds the inevitable exponential state blowup due to interleaving of the traces of parallel modules, worsening the already exceedingly difficult *state-space explosion problem.* As a result, designers prefer to abstract away (ie not deal with) security considerations in the design phase and postpone it to the post-implementation phase.

To provide a realistic real-time model checking experience for the designer even when his models include time and the added complexity of security we pursue an approach inspired by *unit verification* [20] called *security check.* The properties we prove in *security check* are safety properties. We also check something called quasi-liveness or bounded response which is a reasonable weakening of classical liveness. Both these classes of properties are inherent in any security property specification. While safety deals with properties of the form "nothing bad will happen"[the private key can never be revealed] liveness deals with assured-response or in a temporal setting bounded-response[the system will always respond in "t" time units even when under a DOS attack]. *Security check* works by taking the property to be proved on the system and suitably crafting a "test process" based on that property(safety or liveness). The "unit", or modules inside the system to which the property is applicable, is isolated, all the behavior of the process not relevant to the property in question is "sealed" off and this transformed "unit" is first minimized and then run in parallel with the test process. Then we check if the test process terminates by emitting a pre-designated "good" or a "bad" transition. Depending on the transition the test process emitted we can say if the property

New Security Paradigms Workshop 2003 Ascona Switzerland

is satisfied by the system or not.

1.1 Benefits

The immediate benefits of this approach are obvious. Huge state spaces become tractable because only the part of the state space relevant to the property in question is traversed while the uninteresting part of the system is abstracted away by "internalizing" the relevant state transitions.. The conversion of external actions into internal actions also aids in minimizing the system to the furthest extent possible when checking the property in question. This "attack" on decreasing the state space by doing a targeted traversal of the state space leads to a dramatic reduction in the space needed to store the model. Consequently we get real results in real time.

One of the biggest benefits of our approach is that we can make safety guarantees even in the presence of unknown attacker behavior. If a component passes a safety property then, because of the compositional nature of our formalism, we can claim that this component will always pass the safety property no matter what environmental context it is placed in. In other words, we can guarantee that regardless of what an attacker's behavior may be, this safety property can never be violated as any attacker behavior will always be a subset of all possible environment behaviors. This enables us to reason about the vulnerabilities of the system even when possible attacks on the system are not known.

Our approach also provides a model of how a component behaves when it is exposed to to specific pre-determined attacks. In this mode of operation, the tests are no longer automata-encodings of safety/quasi-liveness properties but are attack agents. Attack agents are also automata with "good" and "bad" transitions respectively encoding the failure or the success of the attack. When an attack agent runs in parallel composition with the component under study, it exercises a series of traces inside the component. These traces can be marked as suspicious behavior and used subsequently for intrusion detection [25] .[The labels on the transitions of the component system may be system calls in which cases the exercised traces provide one with suspicious sequences of system calls]

Related Work

Compositional approaches to security property checking have been studied in [11] where the authors present CoSec, a compositional tool for formally analyzing security properties. CoSec also provides support for checking information flow properties like noninterference. However, they do not provide any state-space saving optimizations and consequently their models suffer from the classical problems mentioned previously.

Symbolic model-checking techniques [3] do not construct an explicit representation of the state space and so theoretically are a solution for the state-space-explosion problem. However, the concise representations constructed for symbolic analysis make it difficult to simulate these models. We feel that one of the primary advantages of pre-implementation modeling is the ability to create working simulateable system prototypes. Symbolic approaches force us to sacrifice that functionality. In addition there are added complexities introduced for the symbolic approach if time is being modeled .

Outline

Section 2 deals with the underlying semantic definitions for the system. Section 3 defines *security check* while Section 4 provides conclusions and future work.

2. DISCRETE-TIME LABELED TRANSITION SYSTEMS

The basic semantic framework used in our modeling is *discrete-time labeled transition systems*. To define these we first introduce the following.

DEFINITION 1. *A set A is a set of* visible actions *if is is non-empty and does not contain τ or* 1.

In what follows visible-action sets will correspond to the atomic interactions users will employ to build system models. The distinguished elements τ and 1 correspond to the *internal action* and *clock-tick* (or *idling*) action. For notational convenience, given a visible action set A we define:

$$\begin{aligned} A_{\{\tau\}} &= A \cup \{\tau\} \\ A_{\{1\}} &= A \cup \{1\} \\ A_{\{\tau,1\}} &= A \cup \{\tau, 1\} \end{aligned}$$

We sometimes call the set $A_{\{\tau,1\}}$ an *action set* and $A_{\{\tau\}}$ as a *controllable-action set* (the reason for the latter being that in many settings, actions in this set can be "controlled" to a certain extent by a system environment).

Discrete-time labeled transition systems are defined as follows.

DEFINITION 2. *A* discrete-time labeled transition system *(DTLTS) is a tuple $\langle S, A, \rightarrow, s_I \rangle$ where:*

1. *S is a set of* states;
2. *A is a* visible-action set *(cf. Def. 1);*
3. *$\longrightarrow \subseteq S \times A_{\{\tau,1\}} \times S$ is the* transition relation, *and*
4. *$s_I \in S$ is the* start state.

A DTLTS $\langle S, A, \rightarrow, s_I \rangle$ satisfies the maximal-progress property *if for every s such that $s \xrightarrow{\tau} s'$ some s', $s \not\xrightarrow{1} s''$ for any s''.*

A DTLTS $\langle S, A, \rightarrow, s_I \rangle$ encodes the operational behavior of a real-time system. States may be seen as "configurations" the system may enter, while actions represent interactions with the system's environment that can cause state changes. The transition relation records which state changes may occur: if $\langle s, a, s' \rangle \in \longrightarrow$ then a transition from state s to s' may take place whenever action a is enabled. Generally speaking, τ is always enabled; other actions may require "permission" from the environment in order to be enabled. Also, transitions except those labeled by 1 are assumed to be instantaneous. While unrealistic at a certain level, this assumption is mathematically convenient, and realistic systems, in which all transitions "take time", can be easily modeled. We write $s \xrightarrow{a} s'$ when a system in state s transitions, via action a, to state s'.

If a DTLTS satisfying the maximal progress property is in a state in which internal computation is possible, then no idling (clock ticks) can occur.

DTLTSs model the passage of time and interactions with a system's environment. Discrete-time process algebras such as Temporal CCS [16] enrich the basic theory of DTLTSs with operators for composing individual DTLTSs into systems that may themselves be interpreted via (global) DTLTSs. Such languages typically include operators for parallel composition and action scoping, among others. The variant of Temporal CCS used in this paper, for instance, may be defined as follows. Let Λ be a nonempty set of *labels* not containing τ and 1, and fix $A^{\text{TCCS}} = \Lambda \cup \{\overline{\lambda} \mid \lambda \in \Lambda\}$, where a syntactic operator. Intuitively, Λ contains the set of *communication channels*, with visible Temporal CCS actions of the form λ corresponding to receive actions on port λ and $\overline{\lambda}$ corresponding to send actions on port λ. Then (a subset of) Temporal CCS is the set of

terms defined by the following grammar, where M is a maximal-progress DTLTS whose action set is A^{TCCS} and $L \subseteq \Lambda$.

$$P ::= M \mid P_1|P_2 \mid P\backslash L$$

Intuitively, these constructs may be understood in terms of the communication actions and units of delay (or idling) they may engage in. $P_1|P_2$ represents the parallel composition of P_1 and P_2. For the composite system to idle, both components must be capable of idling. Non-delay transitions are executed in an interleaved fashion; moreover, if either P_1 or P_2 is capable of an output ($\overline{\lambda}$) on a channel λ that the other is capable of an input on (λ), then a synchronization occurs, with both processes performing their actions and a τ resulting: in this case, no idling is possible until after the τ is performed. If $L \subseteq \lambda$ then $P\backslash L$ defines a process in which the channels or actions in L may be thought of as "local". In other words, actions involving the channels in the set L are prevented from interacting the environment outside. The net effect is to "clip", or remove, transitions labeled by such actions from P. Other operators, including a *hiding operator* $P[L]$ that converts actions whose labels are in L into τ actions, may be defined in terms of these.

This informal account may be formalized by giving rules for converting Temporal CCS terms into DTLTSs in the standard Structural Operational Style [18].

Finally, DTLTSs may be *minimized* by merging semantically equivalent but distinct states. In this paper a specific equivalence, Milner's *observational equivalence* [22], is used for this purpose. Intuitively, two states in a DTLTS are observationally equivalent if, whenever one is capable of a transition to a new state, then the other is capable of a sequence of transitions with the same "visible content" to a state that is observationally equivalent to the new state. To define observational equivalence precisely, we use the following notions.

DEFINITION 3. *Let $M = \langle S, A, \rightarrow, s_I \rangle$ be a DTLTS, with $s, s' \in S$ and $a \in A_{\{\tau,1\}}$.*

1. *$s \stackrel{\epsilon}{\Longrightarrow} s'$ if there exists $s = s_0, \ldots s_n = s'$ such that for all $0 \leq i < n$, $s_i \stackrel{\tau}{\longrightarrow} s_{i+1}$.*
2. *$s \stackrel{a}{\Longrightarrow} s'$ if there exists s_1, s_2 such that $s \stackrel{\epsilon}{\Longrightarrow} s_1 \stackrel{a}{\longrightarrow} s_2 \stackrel{\epsilon}{\Longrightarrow} s'$.*
3. *The* visible content, *$\hat{a}$, of a is defined by: $\hat{\tau} = \epsilon$ and $\hat{a} = a$ if $a \neq \tau$.*
4. *A relation $R \subseteq S \times S$ is a* weak bisimulation *if, for every $a \in A_{\{\tau,1\}}$ and $\langle s_1, s_2 \rangle \in R$, the following hold.*
   (a) *If $s_1 \stackrel{a}{\longrightarrow} s_1'$ then there exists s_2' such that $s_2 \stackrel{\hat{a}}{\Longrightarrow} s_2'$ and $\langle s_1', s_2' \rangle \in R$.*
   (b) *If $s_2 \stackrel{a}{\longrightarrow} s_2'$ then there exists s_1' such that $s_1 \stackrel{\hat{a}}{\Longrightarrow} s_1'$ and $\langle s_1', s_2' \rangle \in R$.*
5. *s_1 and s_2 are* observationally equivalent, *written $s_1 \approx s_2$, if there exists a weak bisimulation R with $\langle s_1, s_2 \rangle \in R$.*

Intuitively, $s \stackrel{\epsilon}{\Longrightarrow} s'$ if there is a sequence of internal transitions leading from s to s', while $s \stackrel{a}{\Longrightarrow} s'$ if there is a sequence of transitions, one labeled by a and the rest by τ, leading from s to s'. The visible content of τ is "empty" (ϵ).

It can be shown that observational equivalence is indeed an equivalence relation on states, and that observationally equivalent states in a DTLTS can be merged into single states without affecting the semantics of the over-all DTLTS.[1] It is also the case that, in the context of the Temporal CCS operators mentioned above, DTLTSs may be freely replaced by their minimized counterparts without affecting the semantics of the overall system description. For finite-state DTLTSs, polynomial-time algorithms for minimizing DTLTSs with respect to observational equivalence have been developed [5, 10, 13, 17].

2.1 Model Checking

In automated model-checking approaches to system verification system properties are formulated in a temporal logic; the model checker then determines whether or not they hold of a given (finite-state) system description. The given formula defines a behavior the system should or should not have as it executes. The logic used for expressing formulas contains operators enabling one to describe how a system behaves as time passes rather than simply a characteristic of the system at a particular point in time.

In this work we use a (very small) subset of the *modal mu-calculus* [14], a temporal logic for describing properties of (discrete-time) labeled transition systems. The syntax of the fragment is described as follows, where A is a visible-action set (cf. Def. 1).

$$\phi ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle a \rangle_{\{\tau\}}\phi \mid [a]_{\{\tau\}}\phi \mid \langle a \rangle_{\{\tau,1\}}\phi \mid [a]_{\{\tau,1\}}\phi$$

Here $a \in A_{\{1\}} \cup \{\epsilon\}$. The full mu-calculus contains other operators, including conjunction, disjunction and recursion constructs; a full account may be found in [14].

These formulas are interpreted with respect states in a given DTLTS. The formulas tt and ff represent the constants "true" and "false" and hold of all, respectively no, states. The remaining operators are *modal* in that they refer to the transition behavior of a state. In particular, a state s satisfies $\langle a \rangle_{\{\tau\}}\phi$ if if there is another state s' such that $s \stackrel{a}{\Longrightarrow} s'$ and s' satisfies ϕ, while s satisfies $[a]_{\{\tau\}}\phi$ if every s' such that $s \stackrel{a}{\Longrightarrow} s'$ satisfies ϕ. The operators $\langle a \rangle_{\{\tau,1\}}$ and $[a]_{\{\tau,1\}}$ are similar except that they treat clock ticks as being analogous to τ-transitions. More precisely, we define the following.

DEFINITION 4. *Let $M = \langle S, A, \longrightarrow, s_I \rangle$ be a DTLTS, with $s, s' \in S$ and $a \in A_{\{\tau,1\}}$.*

1. *$s \stackrel{\epsilon}{\Rightarrow} s'$ if there exists $s = s_0, \ldots, s_n = s' (n \geq 0)$ and $a_1, \ldots a_n$ such that $s_0 \stackrel{a_1}{\longrightarrow} s_1 \cdots s_{n-1} \stackrel{a_n}{\longrightarrow} s_n$ and $a_i \in \{\tau, 1\}$ for all $1 \leq i \leq n$.*
2. *$s \stackrel{a}{\Rightarrow} s'$ if there exists s_1, s_2 such that $s \stackrel{\epsilon}{\Rightarrow} s_1 \stackrel{a}{\longrightarrow} s_2 \stackrel{\epsilon}{\Rightarrow} s'$.*

So $s \stackrel{\epsilon}{\Rightarrow} s'$ if there is a sequence of $\tau-$ and $1-$transitions leading from s to s', while $s \stackrel{a}{\Rightarrow} s'$ if there is a sequence of transitions, one labeled by a and the rest either by τ or 1, leading from s to s'.

We can now define $\langle a \rangle_{\{\tau,1\}}$ and $[a]_{\{\tau,1\}}$ more precisely. A state s satisfies $\langle a \rangle_{\{\tau,1\}}\phi$ if there is an s' such that $s \stackrel{a}{\Rightarrow} s'$ and s' satisfies ϕ. Dually, s satisfies $[a]_{\{\tau,1\}}\phi$ if every s' reachable via a $\stackrel{a}{\Rightarrow}$ transition from s satisfies ϕ.

The operators $\langle\ \rangle_{\{\tau\}}$, $[\ ]_{\{\tau\}}$, $\langle\ \rangle_{\{\tau,1\}}$ and $[\ ]_{\{\tau,1\}}$ are not primitive mu-calculus operators, but they can be encoded using the primitive operators.

[1] More precisely, the notion of observational equivalence can be lifted to a relation between DTLTSs, rather than just between states in the same DTLTS. It can then be shown that a DTLTS is observationally equivalent to its minimized counterpart.

In what follows we write $M \models \phi$ if M is a DTLTS whose start state satisfies ϕ.

2.1.1 The Concurrency Workbench of the New Century

We use the Concurrency Workbench of the New Century (CWB-NC) [5, 6, 21] as the verification engine for *security check*.

The CWB-NC supports several different types of verification, including mu-calculus modeling checking, various kinds of refinement checking, and several types of semantic equivalence checking. The tool also includes routines for minimizing systems with respect to different semantic equivalences, including observational equivalence.

The design of the CWB-NC makes it relatively easy to incorporate support for different design languages. The Process Algebra Compiler (PAC) tool [4, 21] provides support for adapting the design language processed by the CWB-NC.

3. SECURITY CHECK

Security check is a specialization of unit verification which in turn derives its name from *unit testing*. In unit testing, software modules are first tested in isolation before being assembled into full systems. In order to test a module that may, in the final system, not have an interface to the external environment, one typically constructs a *test harness* that drives the execution of the software under test. Unit testing is frequently used in software projects because it gives engineers an ability to detect bugs at the module level, when they are easier to diagnose and fix. For unit testing to work, of course, one must have module-level requirements at hand so that test results can be analyzed.

In unit verification, the set-up is very similar to unit testing: single modules are verified in isolation using "harnesses" to provide the stimuli that the other modules in the system (or the external environment) would generate once the module is deployed. As with unit testing, this approach requires the presence module-level requirements so that results can be correctly interpreted.

Security check specializes unit verification by defining tests as automata encodings of security properties. *Security check* also extends the unit verification framework by providing support for attack agents which are then used to extract traces of the module under test. If the module fails the test(ie the attack is successful) then the trace obtained in the module can be used to modify the design to remove the vulnerability. Even when the module passes the test a simulation-based enunciation of the traces of the module can enable the designer to note down the internal behavior of the system when under attack. Once the system has been deployed, this information can later be used for detecting intrusions/attacks on the system.

3.1 Security Properties

A *security policy* defines a system execution that, for one reason or another, has been deemed unacceptable. [24]. Formally a security policy is specified by providing a predicate on sets of executions. A *security property* is a security policy (ie a set of executions) for which set-inclusion of a particular execution into the set of allowed executions can be determined by the execution alone and not by other members (executions) of the set. Information flow properties [26] like non-interference cannot be determined by studying only a particular execution: to deduce if information flows from a variable x to another variable y in a given execution, one has to look at the values y takes in other executions to determine if there exists a correlation between x and y. It has been shown by Schneider et al that security properties can be checked in a compositional manner.

Security check deals primarily with *trace properties*: properties of system executions. In other words, we will be concerned with security properties rather than the more general security policies. This is because our approach is fundamentally compositional in approach and we wish to use the compositional property of security properties. In this section we sketch a basic theory of security properties.

As executions may be thought of as sequences, we use standard mathematical operations on sequences in what follows: if A is a set, then A^* is the set of sequences whose elements come from A, if σ, σ' are sequences then $\sigma \cdot \sigma'$ is the sequence obtained by concatenating them in the given order, ϵ is the empty sequence, etc.

DEFINITION 5. *Let $M = \langle S, A, \longrightarrow, s_I \rangle$ be a DTLTS.*

1. *Let $s, s' \in S$ be states and $\sigma \in (A_{\{1\}})^*$ be a sequence of (non-τ) transition labels. Then*

$$s \stackrel{\sigma}{\Longrightarrow} s' \text{ if } \begin{cases} \sigma = \epsilon \text{ and } s \stackrel{\epsilon}{\Longrightarrow} s' \text{ in Def. 3(1), or} \\ \sigma = a \cdot \sigma' \text{ and } \exists s'' \in S. s \stackrel{a}{\Longrightarrow} s'' \stackrel{\sigma'}{\Longrightarrow} s'. \end{cases}$$

2. *The* language, *$L(M, s)$, of $s \in S$ is defined by:*

$$L(M, s) = \{\sigma \in (A_{\{1\}})^* \mid s \stackrel{\sigma}{\Longrightarrow}' \text{ some } s' \in S\}.$$

3. *The* language, *$L(M)$ of M is defined by:*

$$L(M) = L(M, s_I).$$

The *language* of a state in a DTLTS contains the sequences of visible actions / clock ticks that a user can observe as execution of the DTLTS proceeds from the state. The language of the DTLTS is just the language of the start state.

In this case study the properties we are concerned with involve system executions and come in two varieties: *safety* and *quasi-liveness*. These are defined as follows.

DEFINITION 6. *Let $M = \langle S, A, \longrightarrow, s_I \}$ be a DTLTS.*

1. *A* safety or quasi-liveness property over *A is any subset of $(A_{\{1\}})^*$.*

2. *M satisfies safety property $\mathcal{S}$ iff $L(M) \subseteq \mathcal{S}$.*

3. *M satisfies quasi-liveness property $\mathcal{Q}$ iff for every σ, s such that $s_I \stackrel{\sigma}{\Longrightarrow} s$, there exists $\sigma \in L(M, s)$ such that $\sigma \cdot \sigma' \in \mathcal{Q}$.*

Intuitively, a safety property contains "allowed" execution sequences; a system satisfies such a property if all the system's executions are allowed. A quasi-liveness property is more complicated: it contains sequences that a system, regardless of its current internal state, should be able to complete. We call these properties *quasi-liveness* because the definition of satisfaction does not require that such "complete-able" executions actually be completed, only that the system always be capable of doing so. At first blush, this requirement may not seem strong enough to ensure "liveness" in the tradition sense. However, our intuition is that, if a quasi-liveness property is satisfied by a system, then in any "reasonable" run-time setting employing some kind of fair scheduling, a "complete-able" execution will eventually be completed. These definitions are inspired by, but differ in several respect from, the classic definitions of safety and liveness in [1].

3.1.1 *Defining* Security Check

The approach we advocate may be used to check whether a system satisfies safety / quasi-liveness properties as defined in the previous section. The method consists of the following general steps, where M is the module being analyzed and P is the property.

1. Construct a *security harness* $H_P[\,]$.
2. Plug M into $H_P[\,]$, yielding a new system $H_P[M]$.
3. Apply a check to $H_P[M]$ to see if M satisfies P or not.

The checks applied to $H_P[M]$ depend on whether P is a safety or quasi-liveness property.

In the remainder of this section we flesh out the *security check* approach in the context of Temporal CCS. We define what security harnesses $H_P[\,]$ are and the checks that are applied on $H_P[M]$. We also discuss optimizations to the procedure that can be undertaken to improve (often greatly) performance.

3.2 Security Harnesses in Temporal CCS.

Security harnesses are intended to "focus attention" on interesting execution paths in a module being "security checked". The general form of a security harness is:

$$(V_P \mid [\,])\backslash\Lambda$$

where Λ is the set of all communication labels, V_P is a (deterministic) Temporal CCS expression that we sometimes call a *security process*, and $[\,]$ is the "hole" into which the module to be verified is to be "plugged".

In our setting, security processes (our automata-theoretic encodings of security properties) draw their visible actions from A^{TCCS} (the Temporal CCS action set introduced in Section 2) augmented with two special actions, **good** and **bad**. The latter are used to determine what properties a security process defines. Recalling that the semantics of Temporal CCS specifies how Temporal CCS expressions may be "compiled" into single DTLTSs, in what follows we assume that our verification processes are single DTLTSs.

In order to characterize the properties associated with a security process V, we first note that V is intended to run in parallel with the module being checked. In order to guide the behavior of the module, V must synchronize with the modules actions, meaning that when V wants the module to perform an input action a, V must perform the corresponding output $\overline{a}$. In general, then, since module properties refer to the actions in the module, to associate a module property with V we need to reverse input / output roles in V's execution sequences. To make this precise we introduce the following notation.

DEFINITION 7. *Let $\sigma \in (A^{\mathrm{TCCS}}_{\{1\}})^*$ be a sequence of externally controllable actions. Then $\overline{\sigma} \in (A^{\mathrm{TCCS}}_{\{1\}})^*$ is defined inductively as follows, where $a \in A^{\mathrm{TCCS}}_{\{1\}}$.*

1. $\overline{\epsilon} = \epsilon$
2. $\overline{a \cdot \sigma'} = \overline{a} \cdot \overline{\sigma'}$, *where* $\overline{\overline{\lambda}} = \lambda$ *and* $\overline{1} = 1$.

A security process V defines both a safety property, $\mathcal{S}(V)$, and a quasi-liveness property, $\mathcal{Q}(V)$, as follows.

$$\begin{aligned}\mathcal{S}(V) &= \{\sigma \in (A_{\{1\}})^* \mid \\ &\qquad \nexists \sigma_1, \sigma_2 . \overline{\sigma} = \sigma_1 \cdot \sigma_2 \wedge \sigma_1 \cdot \mathsf{bad} \in L(V)\} \\ \mathcal{Q}(V) &= \{\overline{\sigma} \mid \sigma \in L(V) \wedge \sigma \cdot \mathsf{good} \in L(V)\}\end{aligned}$$

Intuitively, if **bad** is possible as the next action in an execution then the execution, and all possible ways of extending it, are removed from $\mathcal{S}(V)$. Similarly, action sequences leading to the enabling of **good** are included in the property $\mathcal{S}(V)$.

3.2.0.1 *Defining Safety and Quasi-Liveness Checks..*

From the structure of $H_P[\,]$ one can see that the only actions that $H_P[M]$ can perform for any M are τ, 1, **good** and **bad**. This is due to the fact that $H_P[M] = (V_P \mid M)\backslash\Lambda$, and the $\backslash\Lambda$ operator prevents all but these actions from being performed. This fact greatly simplifies the task of checking whether or not a safety / quasi-liveness property encoded within a security process holds of a module.

THEOREM 1. *Let M be a Temporal CCS system model and V be a security process. Then the following hold.*

1. *M satisfies $\mathcal{S}(V)$ iff $(V \mid M)\backslash\Lambda \models [\mathsf{bad}]_{\{\tau,1\}}\mathsf{ff}$*
2. *M satisfies $\mathcal{Q}(V)$ iff $(V \mid M)\backslash\Lambda \models [\epsilon]_{\{\tau,1\}}\langle\mathsf{good}\rangle_{\{\tau,1\}}\mathsf{tt}$*

*proof:*Follows immediately from the definitions of $|$, $\backslash L$, $\mathcal{S}$ and $\mathcal{Q}$. The determinacy of V is important.

This theorem says that the correct "check" for the safety property encoded in a security process V is to see whether or not the "plugged-in" security harness, $(V \mid M)\backslash\Lambda$, forever disables the **bad** action: formula $[\mathsf{bad}]_{\{\tau,1\}}\mathsf{ff}$ holds exactly when there are no execution sequences consisting of τ's, 1's and a single **bad** action. Likewise, to check if V's liveness property holds of M, it suffices to check that $(V \mid M)\backslash\Lambda$ satisfies $[\epsilon]_{\{\tau,1\}}\langle\mathsf{good}\rangle_{\{\tau,1\}}\mathsf{tt}$: if so, then regardless of what M does, there is still a possibility of $(V \mid M)\backslash\Lambda$ evolving to a state in which **good** is enabled.

In some cases, it may be more natural to "look for bugs" rather than to try to prove the nonexistence of bugs. This might be the case if, for example, one strongly suspects erroneous behavior(ie a vulnerability). To determine if a module violates a security process's safety property, one may perform the following check:

$$(V \mid M)\backslash\Lambda \models \langle\mathsf{bad}\rangle_{\{\tau,1\}}\mathsf{tt}$$

If the answer is "yes" then a violation exists. Similarly, one may check

$$(V \mid M)\backslash\Lambda \models \langle\epsilon\rangle_{\{\tau,1\}}[\mathsf{good}]_{\{\tau,1\}}\mathsf{ff}$$

to test whether or not M violates V's quasi-liveness property.

Optimizations

So far our basic methodology consists of the following steps.

1. Formulate a security process V.
2. To check whether or not V's safety / quasi-liveness property holds of M, check whether or not simple modal mu-calculus formulas hold of V "running in parallel with" M.

Two simple optimizations greatly facilitate this process; we describe these here.

Minimization. Checking whether or not a mu-calculus property holds of a system requires, in general, a search of the system's state space. Reducing the size of this state space thus reduces the time required by this search. In the case of $(V \mid M)\backslash\Lambda$, one way to reduce states in the parallel composition is to reduce states in V and M by minimizing them with respect to observational equivalence.

Action Hiding. In a property of the form "the return address cannot be overwritten", actions not related to writing to the address space are unimportant. Mathematically, this is reflected in

the structure of a security process: every state has a self-loop for every unimportant action, since such actions do not "affect" the verification result.

This observation can be exploited to reduce the state space of $(V \mid M)\backslash\Lambda$ even further as follows.

1. Partition Λ into a set I of "interesting" labels and a set $U = \Lambda - I$ of "uninteresting labels."
2. Hide actions involving uninteresting labels in M, creating $M' = M\lfloor U \rfloor$ (and likewise for V, creating V').
3. Minimize M' and V' and perform the safety / quasi-liveness check on $(V' \mid M')\backslash I$.

Hiding actions turns them into τ's; this process enhances possibilities for minimization, since observational equivalence is largely sensitive only to "visible" computation. It should also be noted that TCCS (unlike conventional Statecharts [7] has a compositional semantics. So properties proved on sub-components can be "lifted" to more complex systems in a precise way. This compositional property plus the optimizations detailed above lead to massive state-space reductions. [20] and consequently an easier, real-time analysis effort.

A note of caution is in order here. Hiding actions in Temporal CCS turns them into τ actions. Since Temporal CCS has the *maximal progress property* (cf. Def. 2, introducing cycles of τ's via hiding can cause timing behavior to be suppressed (a τ-cycle can cause "time to stop"). When hiding actions, care must be taken not introduce such loops, or *divergences*, as they are often called. The CWB-NC model checker may be used to check for the presence or absence of divergences.

Putting It All Together

What follows summarizes our general approach to checking safety and quasi-liveness properties with *security check*. To check a safety or quasi-liveness property of a module M:

1. Formulate an appropriate security process V.
2. Identify the interesting (I) and uninteresting (U) labels in M.
3. Form $M' = M\lfloor U \rfloor$, which hides the actions involving uninteresting labels in M. Make sure no divergent behavior is introduced into M'.
4. Minimize M', yielding M''.
5. Do the same on V if necessary, yielding V''.
6. To check V's safety property: determine whether or not $(V'' \mid M'')\backslash I \models [\mathsf{bad}]_{\{\tau,1\}}\mathsf{ff}$.
7. To check V's quasi-liveness property: determine whether or not $(V'' \mid M'')\backslash I \models [\epsilon]_{\{\tau,1\}}\langle \mathsf{good} \rangle_{\{\tau,1\}}\mathsf{tt}$.

3.3 Attack Agents

The other type of analysis that *security check* supports is done with attack agents. Here we lose the generality of safety/liveness properties as now we are dealing with known attacks. But the upside is that we can do some analysis that was not possible in the more general case.

The approach is almost similar to the one illustrated previously except for some differences. The first difference is that here the security harness consists of the attack agent instead of the security process. Secondly we do not hide the interesting internal actions as it is precisely these actions that we want to observe. If a module fails an attack then that means that the "bad" transition is possible in the attack agent.[A "good" transition in an attack agent obviously means that the attack hasn't been successful].Formally that means:
M fails an attack V iff $(V \mid M) \models [A]_{\{\tau,1\}}\langle \mathsf{bad} \rangle_{\{\tau,1\}}\mathsf{tt}$ (where A is the set of all labels of M) .

The intuition behind this is that no matter what externally visible actions M performs in the context of the attack agent, there exists a way by which a "bad" transition can be fired(ie the attack can succeed). Once we obtain such a result, we are naturally interested in knowing what trace inside M is responsible for this failure . This can be done using the search facility of CWB-NC (discussed later). Even if the test returns a "no" ie M is not vulnerable to the attack we can use the CWB-NC simulator to step through the traces in M to check out the transitions exercised by M when put in parallel with the attack agent. Intrusion detection engines [25] work by operating through a learning phase in which they learn the common system call sequences to create a model of "acceptable behavior" and then by flagging a warning whenever a system call occurs that does not fall into the model of acceptable behavior. In our approach, we create models of 'suspicious behavior" ie behaviors exhibited by the system when under attack. This is done in the pre-implementation phase thus obviating the need for a learning phase during implementation. Specifically if we have a model in *security check* whose transitions are labeled by system calls, then "suspicious behavior" would consist of a set of all system call sequences that are exhibited by the system when under attack.

3.4 Tool Support

The CWB-NC tool includes several routines that support the unit verification procedure described above. Primary among these are two different routines for checking whether or not mu-calculus formulas hold of systems. One, the basic model checker, returns a "yes / no" answer quickly. Another, the search utility, searches from the start state of a system for another state satisfying a given property: if one is found, then the simulator is "loaded" with a shortest-possible sequence of execution steps leading from the start state to the state in question. This enables the user to step through the given execution sequence to examine how the found state was reached. In particular, to determine if a module M violates the safety property of security process V, it suffices to search from the start state of $(V \mid M)\backslash\Lambda$ for a state satisfying $\langle \mathsf{bad} \rangle \mathsf{tt}$ (a mu-calculus formula holding of states from which **bad** is immediately enabled). If such a state is found, then the safety property is violated, and the execution sequence loaded into the simulator may be examined to determine why. In the case of quasi-liveness, the same process may be searched for a state satisfying $[\mathsf{good}]_{\{\tau,1\}}\mathsf{ff}$: if such a state exists then the quasi-liveness property is violated. And in the case that M fails an attack we search from the start state of $(V \mid M)$ for a state satisfying [bad]tt This will give a trace to the state which is responsible for the module failing the attack.

The tool also contains a *sort* utility that, given a Temporal CCS system description, returns all the externally controllable (i.e. non-τ) actions the system can performed. The sort command provides a convenient utility for checking whether or not a safety holds: check whether or not the harnessed process's sort contains **bad**. It also may be used to check for violations of quasi-liveness properties: if the harnessed process's sort does not contain **good**, then the property is violated. The latter is only a sufficient condition: just because **good** is in the sort of such a process does not guarantee that the quasi-liveness property is satisfied.

The CWB-NC also includes a routine for minimizing systems

with respect to observational equivalence.

4. FUTURE WORK AND CONCLUSIONS

With respect to future work, we plan to illustrate our approach with some case-studies. While in this paper security processes are monolithic entities, we intend to generalize this concept to a more distributed setting ie a security property may now be encoded by several automata working in conjunction with each other connected to different parts of the system. We also intend to look at ways for modeling security properties in UML. Another interesting avenue of research is to see how to modify this state-space constraintment methodology so as to be able to check general information flow properties like non-interference.

Security check does not seek to replace post implementation static or dynamic analysis. It is obvious that many security flaws creep in during the coding stage and not in the design phase. *Security check* cannot deal with such vulnerabilities as they come later in the software development life-cycle. But there are certain security flaws which arise due to faulty design decisions and these may be remedied by *security check*. In conclusion, this paper make a case for software designers to deal with security issues at the design phase itself by providing them with a methodology for checking security properties in an efficient, practical way.

Acknowledgments

I wish to thank my advisor Dr Rance Cleaveland for guidance and also Rahul Agarwal for his helpful comments .

5. REFERENCES

[1] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[2] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. *SPIN 2000 Workshop on Model Checking of Software*, 2000.

[3] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 2000.

[4] R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. pages 153–173.

[5] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[6] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. pages 394–397.

[7] D.Harel. Statecharts:a visual formalism for complex systems. *Science of Computer Programming,8*, pages 231–274, 1987.

[8] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. *Operating Systems Design and Implementation*, 2002.

[9] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. Lclint: A tool for using specifications to check code. *SIGSOFT Symposium on the Foundations of Software Engineering,*, December 1994.

[10] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1989/1990.

[11] R. Focardi and R. Gorrieri. The compositional security checker:a tool for the verification of information flow security policies. *IEEE Transactions on Software Engineering 23(9):550-571*, September 1997.

[12] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. *ACM Principles Of Programming Languages*, 2002.

[13] P. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[14] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.

[15] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Berlin, 1992.

[16] Faron Moller and Chris Tofts. A temporal calculus of communicating systems. *Proceedings of CONCUR'90*, 1990.

[17] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[18] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[19] A. Pnueli. The temporal logic of programs. In 18^{th} *Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, October/November 1977. IEEE.

[20] Arnab Ray and Rance Cleaveland. Unit verification: The cara experiences. *To be published in Software Tools For Technology Transfer*.

[21] R.Cleaveland and S.Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 41(1):39–47, 2002.

[22] R.Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 1980.

[23] Fred Schneider. A language-based approach to security. informatics: 10 years back, 10 years ahead. *Lecture Notes in Computer Science, Volume 2000 (Reihnard Wilhelm, ed.), Springer-Verlag*, 2000.

[24] Fred Schneider. Enforcable security policies. *ACM Transactions on Information and System Security*, January 2000.

[25] R. Sekar and P. Uppuluri. Synthesizing fast intrusion detection/prevention systems from high-level specifications. *USENIX Security Symposium*, 1999.

[26] F. Simon. Aggregation and separation as non-interference properties. *The Journal Of Computer Security, 1(2):159–188, 1992.*, 1992.