Model-Checking

Model Checking is used to check if a "model" has a certain "property". Model describes the system whose property is to be checked. Both the model and the property can be expressed as Finite State Automata (FSA). The properties are usually *temporal* in nature, i.e., they talk about behaviors that evolve over time. (Here, time is a logical notion, rather than being related in any strict way to notions of real time.) A property automaton's start state specifies the initial state of the system, and the transitions take it to possible next state. The two automata are combined in order to check if the model satisfies the property. Specifically, noting that FSA accept languages, we can construct an automaton that accepts the intersection of the languages accepted by the property and the model automaton. If (a) this intersection automaton accepts any strings at all, and (b) if the property automaton specifies the *negation* of a property **P** of interest to us, then we can say that **P** is *violated* by the model.

Model Checking can be used in various contexts –

- To analyze individual components in software --- in this case, sequential execution of program statements reflects our notion of time. Thus, we can reason about properties of the form "if variable X has a value 5 at line 10, can it have the value 6 at line 20?" Another example discussed in the lecture concerned the order of execution of function calls, e.g., "can the *system()* be invoked following *setuid()?"*
- To analyze concurrent systems. Model Checking has been very successful in the context of concurrent systems because human intuition, although good in the case of individual components of the system, does not work very well in the case of multiple components functioning simultaneously. Model Checking becomes ideal in this scenario.

Properties can be grouped into two categories.

- Safety properties:- Bad things will not happen. For example, a security vulnerability like format string bug can be the "bad" thing and the safety property assures that format string vulnerability is not possible in this code.
- Liveness property:- Good things will happen eventually. Liveness properties are related to availability.

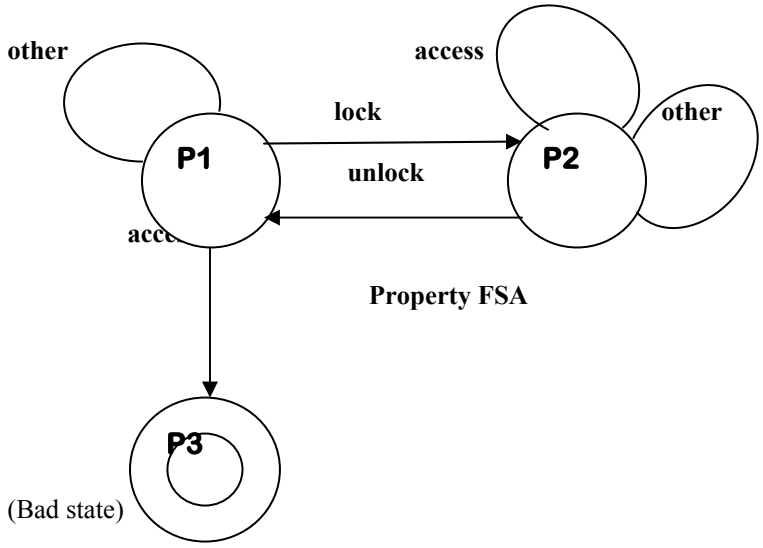
Modeling liveness properties (good things will happen "eventually") is equivalent to modeling an infinite stream of events. This cannot be modeled in Finite State Automata. (FSA can accept strings that are arbitrarily long, but not infinite strings.) Buchi Automata are used to model infinite streams of input. Buchi Automata is essentially FSA but uses annotations that characterize "acceptable cycles," which can be used to characterize properties of infinite streams. (Contrast this with FSA, where acceptance is based on reaching a final state, which implies reaching the end of the stream.) For instance, if all acceptable cycles contain an automaton state where a certain predicate P holds, this means that The cycles contain at least one "good" state so that the state is reached "eventually".

In the context of security, we are more interested in providing safety guarantees rather than availability guarantees (availability is more of a "best effort" thing). Therefore, from now on we will primarily talk about properties concerning safety. As mentioned earlier, we can use FSAs to capture safety properties. In fact, we specify *negations* of safety properties --- the final states of the resulting property automaton corresponds to the "bad states" that we are trying to avoid.

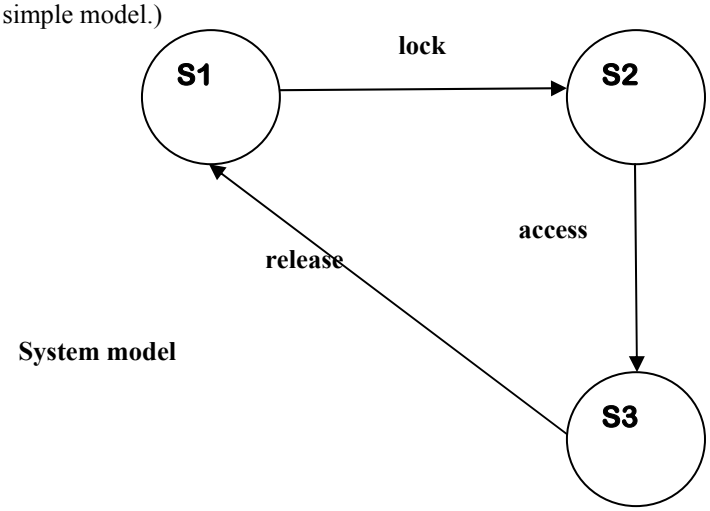
Model checking has been most successful in the context of concurrent systems, where interactions between components lead to bad states. For example, when we are told about a race condition bug in a piece of code, it is hard to figure out when and where the bug is. But a model checker will find a path to the bad state which is causing the fault --- thus finding where the bug is. This is one of the strengths of model-checking, also known as **"Counter example generation"** -- it not only tells you that a property is violated, it gives you a concrete execution sequence (of the model) that leads to this violation.

One of the safety properties of interest relates to permission-checking. In particular, we want to very "complete mediation" in kernel code, i.e., ensure that no matter which path is taken in the kernel to access a resource, we

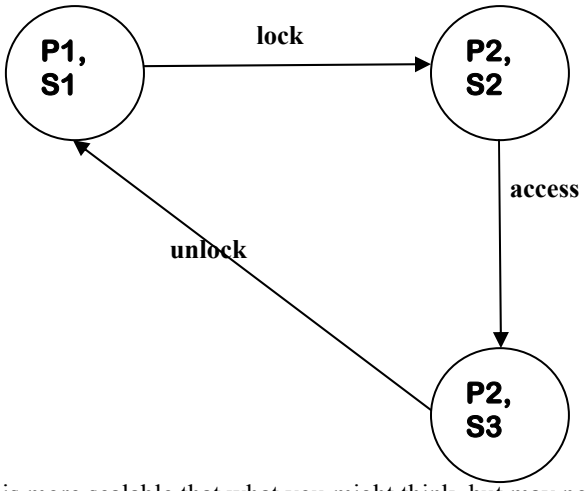
require that the permission checks be performed before the access. (We may also be interested in a weaker version of this property, which enforces this condition only in case of executions initiated by user-level code.) Another version of the same property is one that says a lock must be acquired before accessing a (shared) resource. Let us draw the FSAs to capture the violation of this property. For simplicity, we assume that there is only one resource.



Now, consider the following model. (Usually, the model will represent the behavior of some piece of code, and will hence be much more complex. But since our interest here is in understanding how model-checking works, we will stick to a very simple model.)



In order to determine if the model of the system satisfies the property we need to find compute the product of the property and model FSAs. States in the product FSA are of the form (p, m), where p is a state in the property automaton, and m is a state in the model FSA. A state (p, m) in the product FSA is marked as an accepting state if p is marked as an accepting (i.e., a "bad") state in the property FSA. If the product FSA contains a path to such a final state, then the model violates the property, and this path provides the desired counter-example. The picture below shows the product FSA --- some unreachable states are not shown, e.g., (P1, S3).



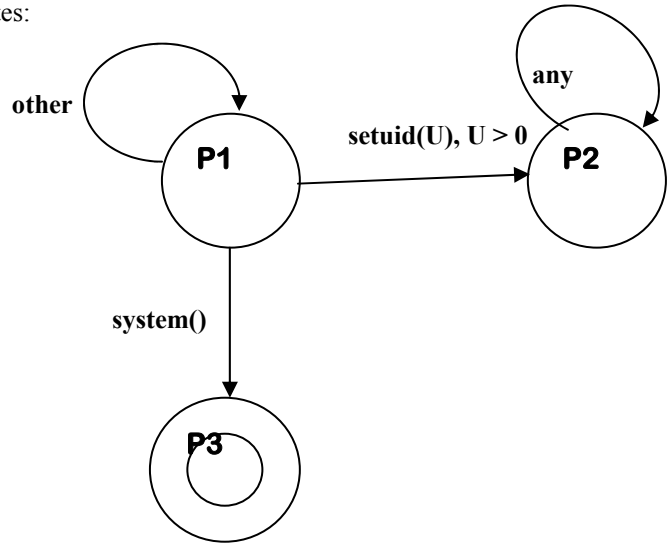
Is this scalable? It is more scalable that what you might think, but may not be enough to handle large software systems, unless the properties are fairly simple. We will discuss more about scalability issues later.

We also looked at another example program in class:

```
loggedin=0;
while (getcmdline(cmd, line)) {
   switch (cmd) {
     case "user":
        if (!loggedin) {
          /* authenticate user, get her uid in variable U */
          setuid(U);
          loggedin=1;
        }
        break;

     case "ls":
        execve("/bin/ls");
        break;

     case "anonymous":
         if (!loggedin) {
            setuid(anon_ftp_uid);
            break;
        }
        break;

     default:
        if ((loggedin) && (isValidCmd(line)))
          system(line);
        break;

   }
}
```

Think of this program as a highly simplified version of an ftp server. It allows remote users to login, and then execute certain commands. Validity of commands is checked by another function isValidCmd that we have omitted, while their actually execution is carried out using the system() function, which creates a shell and hands it the command that is to be executed. Since this program needs to perform user authentication and then change its userid accordingly, this program needs to be started as root. However, for security, we would want to be sure that users cannot execute commands without logging in first. Since we don't want to rely on the correctness of program's internal logic while verifying this property, it would make sense to avoid using variables such as "`loggedin`" in specifying this security property. In particular, noting that setuid() system call with a non-zero argument needs to be used in order to relinquish root privileges (of a process that had root privilege to start with), we might formulate a

property that states that in any execution of this process, it will not execute the system function unless it had previously executed a setuid() operation with a non-zero argument. This can be specified using a simple FSA with 3 states:



To compute the product of this FSA with the model, we first need to convert the program into an FSA. Note that in principle, any running program represents an FSA: it has only a finite amount of storage, and hence the number of possible states of the program is finite. However, in practice, a program that uses 1MB has a state space of $256^{1000000}$ states! Clearly, it is infeasible to enumerate the states or paths in such as large FSA. So, it is necessary to perform some abstractions to reduce the size of the model FSA.

The simplest approach for abstracting the model is to ignore all data values, while retaining control flows in the program. In particular, you create one state in the FSA for each program statement, with transitions capturing the normal flow of control from one program statement to the next, as well as those control-flow transfers that take place as a result of if-then-else, jumps and calls. These transitions are annotated with any operations performed by the program at a given state that is of interest in the specification FSA. Clearly, we can further abstract such an FSA by omitting states that do not perform operations used in the specification.

If we employ the abstraction that eliminates all data values in the above program, we will end up with an FSA that will make any sequence of setuid() and system() calls, and hence would violate the specification. But this is a case of false positives induced by an overly general abstraction. If we use a more refined abstraction that retains some variables (say, `loggedin`) then we will end up with an FSA that has two important states: the first one corresponds to the case where `loggedin` is 0, with a transition on setuid() operation to another state that has `loggedin` set to 1. A transition on system() is present in the second state but not the first, and hence the intersection of this FSA with the specification FSA contains to path to a state corresponding to P3, thus establishing the absence of a vulnerability. (A careful reader will see that this is not entirely correct: in fact, there is a transition from the first to second state on setuid(), whereas the specification requires the argument to this setuid to be nonzero. So, actually what happens is that you still get a false positive corresponding to a case where the root user logs into FTP server. This false positive can be recognized by a programmer and eliminated manually.)

The above discussion shows that one of the key challenges is that of limiting state space by choosing appropriate subset of variables in the model. This is one of the problems that has received a lot of research attention, and continues to do so. With time, we expect larger and larger programs to be handled by model checkers without significant need for human expertise in "tweaking" the abstractions to be used in the model.

Configuration vulnerabilities

Model checking is good when we need to reason about complex interactions in systems, such as those involving concurrent processes, and interactions between input values, configurations and program paths taken. Let us consider some examples in the context of configuration vulnerabilities involving multiple processes.

Example vulnerability

This is another of those vulnerabilities that we will not find in existing systems. In one of the earlier versions of UNIX, there used to be the following programs –

- *Mail:* The "Mail" program delivers mails to users' mailboxes. A user's mailbox was a file owned by the user and delivering an email was nothing but writing to the end of the file. Since the file is owned by the mail recipient whereas the Mail program is invoked by the sender, the Mail program needs to be a setuid program in order to have enough privileges to write to a file owned by the recipient.
- *Comsat :* The Comsat program monitors the user's mailbox and prints the first two lines of the message body (together with information about the sender) on the terminal of the recipient.
- *Login :* The login program keeps track of what terminal the user is logged in. It stores login info in a file called /etc/wtmp. The login program has root privileges because it is has to authenticate the user and pass privilege on to the user. The way the system was setup - /etc/wtmp was world-writable. (Why? This could be a configuration error. Or, no one at the time thought that this can be used to attack the system because someone might have assumed that the contents of the file are not that important and hence need not be protected.)

What are the various attacks that are possible on this system?

- Change the /etc/wtmp such that your name corresponds to someone else's terminal. This will enable you to receive the first two lines of the mails received by that person. This might not be entirely useful even though this is a valid attack that is possible.
- Change the /etc/wtmp file to state that you are logged in on the terminal "/etc/passwd". (Obviously, this is not a terminal, but the comsat program did not care – the terminal was simply another file name, and it wrote to that file.) Now, send yourself an email whose first line is of the form "xyz::0:0:". Comsat now adds this line to /etc/passwd. The effect of this line is to create a new username "xyz" whose userid is 0 (i.e., this is a superuser) and who has an empty password. (If there was a password, then its encrypted version would have appeared between the two consecutive colons.) After sending this email to yourself, you can login as xyz and now have root privileges!

Here, the interactions between the various processes are easy enough to understand after the fact, but given just the description of the three programs above, it one may not realize that such an attack is possible.

You can find more details and examples on using model-checking to discover configuration vulnerabilities in this paper: Model-Based Analysis of Configuration Vulnerabilities. Here is the basic idea. You can model each of the above three programs. In addition, you need to model the OS at some level, so that things such as file permission checking are properly captured. Finally, the attacker (and/or normal user) behavior has to be modeled.

Since we don't know what the attacker would do, we will develop a very general model that is highly non-deterministic. It will say that the user will pick one of the possible actions he can do, and carry it out. This process will be repeated for ever, thus building a sequence of actions. Possible actions in the above model include invocation of Mail program with a nondeterministically chosen set of arguments (i.e., both the recipient and the message body can be arbitrary), logging in, logging out, and reading or writing a nondeterministically chosen file.

We then need to specify a safety property. Let us say we want to identify if it is possible for a non-root user to acquire root privileges. Since such a user would be able to write the password file, it is clear that, given the above model of attacker behavior, there will be a path to a state where the password file has been overwritten, provided there is a path that yields superuser capabilities to the attacker. Thus, the safety property can be one that simply states that the password file has been overwritten. (For the moment, for simplicity, we will assume that there is no legitimate way to change the password file.)

Given the above model, an appropriately coded model-checker can generate the above attack automatically! The "counter-example generation" capability of a model checker translates, in this case, to *automatic generation of attack* to exploit a vulnerability! (You can find the details in the above paper.)

The same basic concept has been used for determining if network vulnerabilities are exploitable. In particular, you may have a network that you administer. Due to practical problems, some of the machines in the network may have certain unpatched vulnerabilities. When viewed individually, it may seem that they are not exploitable, e.g., a web server may be vulnerable to a buffer overflow attack, but you may decide this is not a serious threat since it is firewalled from the Internet. But it may be possible to mount a two-step attack (or multi-step attack, for that matter) where the attacker first compromises a different host within the network, and then uses that host to attack the internal web server. The concept of "attack graphs" has been developed to capture such multi-stage attacks, and the entire area remains one of active research.

State space explosion is the biggest challenge in using model-checking. We alluded to this already in the context of a single program. Note that in a multi-component system, each component will be described using a

finite-state model. The composite behavior of *N* components is given by an FSA that simultaneously captures the behavior of all components. Typically, we are interested in an interleaving semantics, i.e., each transition of the composite FSA corresponds to a transition of one of the components FSAs. Thus, the states in the composite FSA are of the form (s_1, s_2, …, s_N), where s_i is one of the states in the FSA model of the ith component. There is a transition from this state to another state (s'_1,…,s'_N) on an action *A* if and only if there is a 1 <= j <= N such that the model of the *j*th component contains a transition from its state s_j to s'_j on *A*, and for all i <> j, s_i = s'_i. (Basically, the jth component made a transition, while the other components did not.) Note that the number of states in the composite automaton increases exponentially with *N*.

Another important reason for state explosion is the need to capture data values. In particular, for programs that don't use any data, their state is captured purely by their current execution point (basically, the program counter (PC) value). We can map each of the PC values to one of the states of the FSA. However, if the program starts using data values, then an FSA based formulation needs a lot of states to capture this. Suppose that a program uses a 32-bit integer value. Since this integer can have 2^32 possible values, you may need that many states in an FSA that accurately models the value of this variable. Obviously, programs use much more data, so there is a massive explosion in terms of the state space required. As a result, explicit state model checking where the states are enumerated becomes impractical very quickly. **Symbolic model checking** techniques tend to do better, since they don't attempt to explicitly construct huge FSAs. Instead, they use symbolic representations that implicitly represent many different states of the FSA. There are many ways to achieve this. One of the ways is to use constraints to represent a state. So a constraint on two variables x and y, (x > 5) || (y<=5) represents a number of states that satisfy these two conditions without explicitly representing them. The downside of symbolic representation is that reasoning about groups of states is harder than that of reasoning about individual states. So, it becomes a question of finding appropriate representations that provide a good tradeoff between compactness of representation and ease of reasoning.

Summary of difficulties of model-checking

- Efficiency / Scalability: We have already discussed the state space explosion problem that is inherent in the model checking techniques.

- Specification development: For many years, the scalability problems were very severe. As a result, it was necessary for human experts to develop highly abstract models of systems to be verified so that the state-space of the model was small enough to make model-checking practical. However, there is no way to ensure that the expert-chosen abstractions captured all the relevant aspects of system behavior that needed to be verified. It is all too common, especially in contexts such as security, where the expert abstracts out features that are important, and hence the system ends up violating a property even though the model-checking phase indicated that the property held on the system model.

  As the scalability improved, more and more of the model generation phase is being automated. Model generators typically apply abstraction transformations on programs to obtain models. The problem, in terms of choosing the right abstraction, remains. But at least the manual effort in model development and the likelihood of introducing modeling errors (other than those that arise due to the choice of abstraction) are reduced.

    - Someone has to develop the properties. Since properties define the notion of correctness, and since only humans can define the notion of correctness (based on intended function of the system), human effort is needed. Given that vendors do not make the effort needed to fix relatively simple problems such as buffer overflow bugs which are fairly easy to identify, it seems unlikely that they will be able to spend the time needed to specify correctness properties.

      In the last few years, methods were developed to "infer" intended correctness properties automatically from code. One way to do this is to analyze code, and identify properties that seem to hold on most parts of the program, e.g., one can theorize from the kernel code that resource access operations must be preceded by permission checking operations. A programmer (or systems engineer) can indicate whether this is a reasonable specification or not. Subsequently, a

verifier can be used to check if the property holds on all program paths, and generate warnings when this is not true.

Another approach that has been popular in security is based on the observation that many vulnerabilities are generic, e.g., buffer overflows, race conditions, failure to check permissions, etc. As a result, these properties can be defined once by an expert, and the properties can then be checked against any program without requiring the developer of that program to come up with specifications. This is currently done in some software tools. Examples of companies that use these techniques in real applications are Coverity (for C, C++ code) and Fortify Software (for Java).