A Stochastic Model for Intrusions

Robert P. Goldman

Smart Information Flow Technologies (SIFT), LLC
2119 Oliver Avenue South
Minneapolis, MN 55405-2440 USA
rpgoldman@sift.info

Abstract. We describe a computer network attack model with two novel features: it uses a very flexible action representation, the *situation calculus* and *goal-directed* procedure invocation to simulate intelligent, reactive attackers. Using the situation calculus, our simulator can *project* the results actions with complex preconditions and context-dependent effects. We have extended the Golog situation calculus programming with *goal-directed* procedure invocation. With goal-directed invocation one can express attacker plans like "first attain root privilege on a host trusted by the target, and then exploit the trust relationship to escalate privilege on the target." Our simulated attackers choose among methods that can achieve goals, and react to failures appropriately, by persistence, choosing alternate means of goal achievement, and/or abandoning goals. We have designed a stochastic attack simulator and built enough of its components to simulate goal-directed attack on a network.

1 Introduction

In order to best develop the techniques of cyber defense, we must be able to explore them in a scientific way. Practically speaking, we must be able to test cyber defense components without having to bring on-line special network configurations and then conducting possibly very harmful actions on those networks. While it is not a substitute for actual experimentation (both in laboratories, honeypots and on-line), we must be able to simulate internet attacks based on analytic models, just the way the military must be able to simulate conflicts in the real world.

Our stochastic model design comprises multiple, modular, components that will allow security researchers and wargamers to repeatably exercise their sensors and defenses against goal-directed attackers. The core of our stochastic model is built upon an expressive representation of actions, the situation calculus [21, 23], permitting us to accurately model complex cyberworld processes. We have augmented the situation calculus, and its process-modeling extension, Golog [20, 13, 23], with facilities for goal-directed procedure invocation. These permit convenient specification of cyber attackers' plans in terms of the goals they are to achieve, providing modularity in modeling and permitting us to experiment with a wide variety of attack techniques.

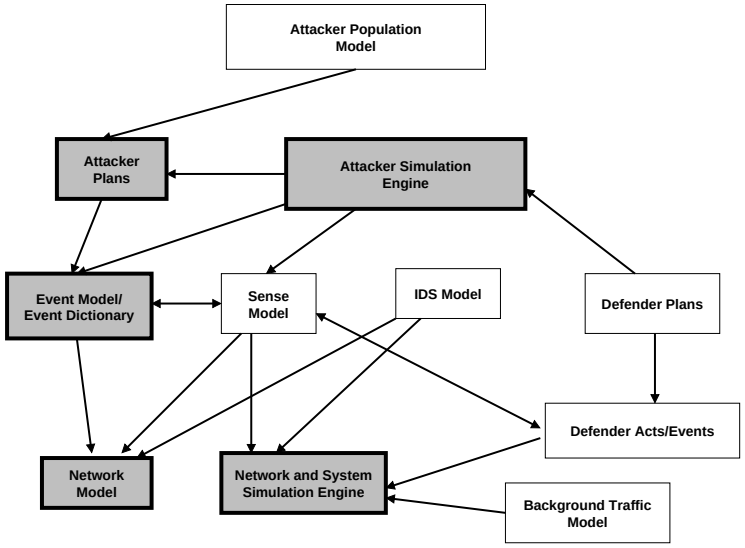

Fig. 1. Illustration of the stochastic model components. Note that the arrows are loosely intended to denote "depends-on" relationships. These are *not* indications of interfaces. The shaded, heavily-outlined components are the ones present in our current prototype.

In the next section, we present the architecture we propose for a full cyber warfare simulation. We have developed a proof-of-concept implementation of the components of that architecture required to simulate a goal-directed attacker. Discussion of that implementation and its theoretical background consume the rest of the paper. We present a full example of a simulated attack, and explain how it was generated by the simulator. Finally, we give proposed future directions and an account of related work.

2 Architecture

We have designed the architecture for a full network warfare model (see Figure 1). We have not implemented this entire architecture; we hope to do so in future work. What we have now is a subset of this architecture sufficient to simulate the actions of a single attacker. We have chosen to focus on attacker simulation because this provides the greatest theoretical and modeling challenges. The current proof-of-concept shows how a single attacker, with a library of plans and their component exploits, can attack a network.

Attacker population model In order to determine how a cyber defense will react "in the wild," we need to be able to consider how it will react to a wide variety of possible attackers. One question of great concern at present is whether we can effectively defend against serious threats like organized crime and terrorists,

without having all our resources absorbed in overreaction against nuisances. This raises the interesting question of whether attackers' intent can be inferred from their actions [10].

We propose to meet this need with a simple stochastic generator that is capable of randomly creating a set of attackers for a scenario. For particular scenarios, and for "wargaming," this could be replaced by having a user just state how many, and of what type, the attackers are, or some combination (randomly generate ankle-biters operating concurrently with a user-specified or even user-controlled team of more serious adversaries). Attackers would be drawn from a set of n types of attacker, $t_1 \ldots t_n$. These might include, for example: script-kiddie, cyber warrior, recreational hacker and cyber terrorist. To experiment with different threat conditions, the probability of appearance of each class could be varied.

Associated with each of these classes of attacker, we should have distributions that determine the attacker's objectives and knowledge. The attacker's objectives would be captured in terms of goals achievable by the plan library. It's not clear what an appropriate set of candidate goals would be, but some obvious choices include:

1. denial of service — for various services run at the site;
2. compromise integrity of service — for various services run at the site;[1]
3. compromise confidentiality of some data at the site;
4. compromise integrity of some data at the site;
5. web site defacement;
6. compromise of root access;
7. springboard for attack on other sites.

Given a finite set of possible goals, $\mathcal{G}$, then for any given attacker, a, who has a set of goals, $G(a)$ and goal, $g \in \mathcal{G}$, there is a set of probabilities as follows: $P(g \in G(a))$, which is a function of the type of the attacker, $t(a)$:

$$t(a_i) = t(a_j) \implies P(g \in G(a_i)) = P(g \in G(a_j))$$

As far as knowledge, each agent will know only a subset of the plans in our entire plan library. Much of the differences in knowledge will be in the specifics of the plan library. That is, overall how to do cracking attacks is pretty well-defined. But down below, when we get to particular exploits on particular services, skill levels get to be radically different. Further, each attacker will have only limited knowledge of the target systems.

Attacker simulation engine Most of the work to date on the stochastic model has centered around this component and the attacker action model, and most of this paper will concern itself with those aspects of the work. We have developed an extension to the Golog logical representation of procedures. That extension adds

[1] Defacement of web site *could* be categorized as compromising the integrity of the service, but that's probably not appropriate, because it's significant as a special kind of attack.

goal-directed invocation facilities, loosely based on reactive planning languages such as PRS [11, 12] and RAPS [9]. The attacker simulation engine is intimately intwined with the plan library and attacker actions.

Attacker plan library The attacker plan library is made up not of specific exploits, but sub-goaling data structures that capture abstract strategies for achieving attack goals. For example, to attain access to a host, you can attain access to a trusted host and then exploit the trust relationship. These would correspond to the tops of attack trees [26], or the most abstract capabilities in JIGSAW [29]. We have developed a collection of simple attacker plans, primarily based on the "Frostbite Falls" scenario [15]. We discuss these plans in more detail in Section 4.1.

Note that attack plans are modeled separately from the attackers' individual actions and from background network traffic (the next two items on our list). There are three reasons to make this separation. First, attacker plans need little, if any, direct interaction with the world model. Second, factoring out the model of individual actions would allow us to experiment with attacker plan recognition [10] without involving the full weight of a simulation engine and a sensor model. Finally, we separate out the details of individual actions because those are the aspects of the model that change most rapidly. Avoiding mention of specific exploits makes it easier to maintain the plan library.

Why does the attacker model not need direct interaction with the world model? Any information consumed (sensing) would come from the model of the attackers' beliefs contained in the state of the attacker simulation engine. That set of beliefs would be populated as a side effect of executing actions. Any changes to the world model would be made through the execution of actions (for which see below).

Attacker actions (Event Model) The attacker actions are modeled using the situation calculus [23]. Attacker actions include not only exploits *per se*, but also conventional actions (e.g., writing a `.rhosts` file), that are used in attacks. The actions provide the bridge between the attacker's plans and the world model.

We model attacker actions in terms of their preconditions and effects. The situation calculus representation provides a crisp semantics for precondition-effects modeling of attacker actions [4, 29]. The attackers' actions are invoked in the plans executed by the attacker simulation engine. Ideally these actions would also be executable in the real world, permitting us to replicate our simulation results in real networks. The model of attacker events is the most developed part of the current system. We have a small but substantial library of exploit actions.

Network simulation At the moment, we have only a relatively static model that describes the current configuration of the network. The network is a passive field upon which the attackers' actions are played out. Eventually we would like to see the network simulation grow to a timed discrete-event model of a computer network. This model must be able to be affected by the invocation of attacker actions in the Event model. It must also provide some kind of time signal that

can be used by other components (especially the attacker simulator so that time is handled correctly. Finally, it must be possible for simulated agents to sense this model and update their beliefs.

Ideally, the network simulation should be plug-compatible with the real world. I.e., we should be able to invoke the attack events (or at least a subset thereof) against the real world to replicate simulation results. We should also be able to have the attacker engine sense the real world and build beliefs the same way it would sense the network simulation.

Sensor Models The proposed sensor model component comprises models of intrusion detectors and other components (e.g. system loggers, firewalls, etc.) that would provide information to a defender. We are particularly interested in developing this component in order to be able to experiment with IDS report aggregation without having to field large suites of IDSes. Such a model would let us do large numbers of experiments on sensor event fusion without having to handle actual traffic, just alerts.

3 Framework for the Single-attacker Simulation

To successfully simulate a computer intruder, we must be able to model primitive actions and their composition. This includes being able to model the actions that are available to the attacker, and their effects on the world (in this case the computer network). The model must also provide a simulated attacker that will intelligently assemble primitive actions into a plan to achieve its objectives. Assembling plans together must include the ability to react to failed actions, either by trying again, choosing new methods that will achieve the same end, or abandoning failed goals.

We have assembled a theoretical framework that meets these needs, and developed a proof-of-concept implementation. We use the *situation calculus* [21, 23] as the framework within which to model attacker actions. The situation calculus provides an expressive framework for encoding actions, including those whose effects are complex functions of the state in which they are executed. Further, the situation calculus includes solutions to the frame and ramification problems, deep problems about reasoning about actions. Golog (ALGOL IN LOGIC) [20] and its variants, Congolog [13] and Indigolog [14] provide ways to express complex procedures in terms that are consistent with the situation calculus. They thus let us describe complex actions, composed of primitive actions, conditionals, etc., in a way that preserves the crisp semantics of the situation calculus. We have augmented the Indigolog framework with *goal-directed invocation*, the ability to invoke procedures based on desired effect, rather than by name. This adds a crucial kind of abstraction to our plan models. We have added stochastic goal persistence and abandonment, so that our agents will react appropriately to failures in their plans. Finally, we have modified an existing Indigolog interpreter [19] to provide the simulation engine for our work.

3.1 The Situation Calculus

As the most mature representation for actions and dynamic change, the situation calculus provides the best framework for us to experiment with descriptions of cyber attacker's actions. The situation calculus is a formal representation for dynamic world models developed by artificial intelligence researchers. The situation calculus is a dialect of first order logic, with certain special features for representing dynamic change.[2] Situation calculus researchers have provided solutions for many knotty problems in representing dynamic change, including management the notorious frame, ramification, and qualification problems.

Recent work on attack modeling by Templeton and Levitt [29] and Cuppens and Ortalo [4] has argued for a "requires-provides" or "precondition-postcondition" model of attack actions. We were inspired by this work, and came to choose the situation calculus framework because it provided a semantics for the pre- and postconditions not yet provided by other work.

Fluents and Predicates The situation calculus is a dialect of typed first order logic. For the purposes of this paper, we assume familiarity with the syntax and semantics of first order logic. The most important distinguishing feature of the situation calculus is the addition of a distinct type, the *situation*. A situation is a snapshot of the world state, together with the history that led to that state. The world is described in terms of static predicates and *fluents*. Static predicates are those that don't change, no matter what actions are taken. Fluents, on the other hand, are predicates that can vary over time, and thus must take situations as arguments. So, for example, *mortal(socrates)* is a formula with a static predicate. On the other hand, *loggedin(b0ri5, host123, s)* tells us that the cracker "b0ri5" is logged into host123 in situation s; *loggedin* is a fluent.

Doing actions In order to reason about the effects of actions, we need to be able to refer to the situation that results from the execution of an action. This is done using the *do* function. *do(a, s)* is a function that denotes the situation that results from doing action a in situation s. In its simplest form, the situation calculus assumes that actions are only executed sequentially, and that all actions are deterministic. There are a number of extensions that permit stochastic actions, concurrent actions (possibly by multiple agents) and reasoning about actions of varying duration.

Projecting the results of actions In order to reason about dynamically-evolving situations, the situation calculus requires axioms of three sorts. The first are those that state the conditions under which it is possible to execute an action. For example,

$$Poss(login(user, host), s) \equiv atconsole(user, host, s)$$

[2] Actually, in part of the formulation of the situation calculus, some second-order quantification is resorted to. However, anyone *using* the situation calculus, will find him/herself on familiar ground.

gives a simple model for when it is possible for a user to log into a host.[3] The second are *successor state axioms* for the fluents. For example,

$$Poss(a, s) \supset [\ loggedin(user, host, (a, s)) \equiv \{\ a = login(user, host) \vee (loggedin(user, host, s) \wedge a \neq logout(user, host))\ \}\ ]$$

That is, if a new situation results from the execution of action a in situation s, then *user* is logged into *host* in the resulting state if and only if either (1) the action a is one of logging into *host* or (2) *user* is logged into *host* in the previous state, and the new action is not one of *user* logging out of *host*. While successor state axioms can be difficult to formulate manually, compilation techniques are available for deriving such axioms from sets of easier to manipulate action descriptions and some closure assumptions [17, 22, 27]. Likewise, when the action descriptions are of limited form, one may simply reason with them directly, without taking the intermediate step of explicitly formulating the successor state axioms. This is the technique adopted by the Indigolog interpreter [19] that we have used.

3.2 Golog and Its Variants

While the situation calculus allows us to reason about the execution of primitive actions, and even sequences of such actions, it is not sufficient to express scripts, or programs, involving looping, conditional execution, etc. The languages Golog [20], and its extension Congolog (Concurrent Golog) [13], were developed to meet this need. Congolog adds control constructs such as branching, conditional execution, etc., and concurrency constructs, to the situation calculus. Using Congolog, one can express a concurrent, branching program, whose atomic actions are described in the situation calculus. This makes it possible to describe a reactive program's interaction with a dynamically-evolving environment. We use these facilities to develop a script-based model of cyber attackers.

Golog allows actions to be combined into programs using the following constructs in Table 1: test, sequence, and nondeterministic choice of action, nondeterministic choice of action arguments and nondeterministic iteration. These constructs should be relatively familiar to those familiar with dynamic logic. Golog also permits procedures (really macros) to be defined (**proc**) and used.

For example, an attacker might want to use a login procedure like the following:

proc $login(host)$
  begin
    if $console\_access(host)$
      then
        $(\pi uid)?(known\_uid(uid, host))$; $(\pi sess)login(host, uid, sess)$

[3] Here, as elsewhere, free variables are implicitly universally quantified.

Table 1. Golog control constructs.

Construct	Notation
Simple actions	a
Test actions	$?\phi$
Sequence	$\delta_1;\ \delta_2$
Nondeterministic choice	$\delta_1 \mid \delta_2$
Nondeterministic choice of argument	$(\pi x)\delta(x)$
Nondeterministic iteration	δ^*

fi
end

The attacker wishes to use this to login to some argument *Host*. First, we check to determine whether the attacker has console access. If not, the procedure simply ends. If the attacker does have console access, then s/he chooses a known uid. Note the use of the nondeterministic choice of argument operator to bind the *uid* argument. Then the attacker completes the procedure by executing the login primitive action.[4] In this case, the environment will, effectively, bind the *sess* (session) argument.

Congolog adds a number of constructs to the above: **if-then-else**, **while** loops, concurrent execution, prioritized concurrency, iterated concurrency and interrupt. See Table 2. Concurrent execution and monitoring will be critical for our modeling of a goal-directed attacker's behavior. For example, a goal-directed attacker determined to shut down a target host might begin activating a number of DDoS servers she "owns," until such a time as she determines that the target host no longer responds to a ping message. We will discuss this further below. Likewise, in order to execute an "ip spoofing" attack, an attacker will concurrently attempt a denial of service attack on the host to be spoofed, while sending forged packets to the target host. This might be encoded as follows:

proc $ip\_spoof(host)$
  begin
    $(\pi t)?(trusted(host,t));\ DoS(t) \parallel spoof\_to(host,t)$
  end

3.3 Our Golog Extensions

The use of the Congolog framework provides us with two key components to our attacker simulation. The first is a representational scheme for actions with complex pre- and post-conditions, and the second is a representation for complex

[4] This is somewhat confusingly given the same name as the procedure. Names are disambiguated by arity.

Table 2. Congolog control constructs.

Construct	Notation
Conditional	**if** ϕ **then** δ_1 **else** δ_2
Looping	**while** ϕ **do** δ_1
Concurrent execution	$(\delta_1 \parallel \delta_2)$
Prioritized concurrency	$(\delta_1 \gg \delta_2)$
Iterated concurrency	$\delta_1^{\parallel}$
Interrupt	$\phi \rightarrow \delta$

attack scripts or plans. However, in its "off the shelf" form, Congolog does not meet all of our needs. First, the semantics of Golog and Congolog rely on "angelic" nondeterminism. To overcome this problem, we have been working with an interpreter for Indigolog. Second, the Golog procedure mechanism is not sufficient to express goal-directed procedure invocation. Recall that we want our simulated attackers to assemble and modify their own attack procedures, based on their goals and the context, and using the actions and procedures available to them. We have developed a framework that adds goal-directed procedure invocation to Indigolog, along the lines provided by PRS [11, 12].

The first concern in designing Golog was to come up with a language for AI agents that would provide clear semantics for plans, and that would allow agents to search for the right plan for a situation. In this case, an appeal to angelic nondeterminism may be appropriate. An agent that is looking for the right way to achieve its goal can use search to explore the possible deterministic sequences of actions that correspond to a given nondeterministic program. The nondeterministic program then provides a convenient shorthand to describe the problem of choosing the right course of action.

However, in situations where the world is not under the complete control of an agent, angelic nondeterminism is not an appropriate construct. For example, there may not be enough information at the point of nondeterministic decision for the agent to avoid painting itself into a corner. Consider a cracker that wants to break into host h by finding a vulnerable host h' that is trusted by h, and achieving root privilege on h'. There is no way for our cracker to know that she can actually achieve root privilege on h' before she tries (although there may be positive and negative indications), and if she tries and fails, she cannot simply backtrack to the state of the world before she has attempted the crack. Many times an alert security analyst will be able to mitigate or stave off a network attack by if she notices evidence of failed attacks. Certainly it would be harder to catch crackers, if they could miraculously undo all the effects of their failed attempts!

There have been a number of efforts to bring together Golog constructs with the needs of robotics control applications, which present these same problems [14, 16]. We have based our work on the Legolog interpreter [19], based

on the Indigolog (INcremental Deterministic Golog) dialect of Golog. This Indigolog interpreter commits to particular strategies for interleaving concurrent actions, choosing between nondeterministic alternatives, etc. The interpreter's functioning is very easy to understand, because it is written in Prolog and directly implements the semantics of Indigolog as described in [14]. However, we will see that some enhancements were necessary to permit goal-driven execution.

In order to build flexible plans for our attackers, we need to be able to use goal-directed invocation or subgoaling. That is, we need to be able to specify parts of the attacker's plan by specifying *what* it is to achieve, instead of how it to do the job. For example, we should be able to specify that a plan requires the attacker to achieve root privilege on a specific Unix host, without specifying *how* the attacker is to do this. There are two reasons we would like to have this feature. The first is a simple matter of software engineering. We would like to be able to write procedures with straightforward interfaces, and the goal of the procedure is a good way to specify the its interface. The goal provides a stronger, more standard way of encoding the interface than conventional name and parameter methods, because unlike those methods, the goal expression has its own semantics. Further, using the goal as interface specification allows us to permit multiple, alternative, methods for the same goal. Finally, if procedures are characterized by their goals, our agents have an operational method for determining whether or not a procedure invocation has succeeded. In turn, this permits them to react appropriately to success and failure.

While we cannot develop a psychologically accurate model of a cyber attacker, we do want to develop a model that is capable of reasoned attack on a defended network. To do so, we have been broadly guided by the "beliefs-desires-intentions" model [3]. As the name suggests, this model proposes that agents have beliefs and desires about the world. Based on those beliefs and desires, the agents adopt goals, and intentions (plans) to pursue those goals. We have also been guided by the design of reactive programming languages like PRS [11, 12] and RAPS [9].

The facility we need is the ability for our agents to pursue goals. Our agents should be able to choose methods to achieve their goals. They should be able to adopt sub-goals as part of a plan to achieve another goal. For example, our agents must be able to decide to take over one host as a stepping stone to a true target. Agents must persist in their goals, and choose alternative methods when one method fails. For example, if an agent is unable to use a specific exploit successfully against a host, that agent might try a number of alternatives: she might try the exploit again (some exploits, like race conditions, are not reliable); she might try a different exploit or she might try to attack a different host. Finally, sub-goals should not last too long. For example, if an agent is performing a denial-of-service attack as part of an ip-spoofing attack, the denial-of-service should not persist past the end of the ip-spoofing attack.

To meet these objectives, we significantly expanded the Indigolog language, adding goal-directed procedures, called KAs (after the similar PRS construct). A KA is like a normal Indigolog procedure, but has additional special features

KA *user_to_root*(*Host*)
  begin
    π *sess*
    ?(*logged_into*(*Host*, *sess*));
    achieve_goal(*root_privileged*(*sess*))
  to achieve
  root_privileged_on(*Host*)
  when
  logged_into(*Host*)
  end

Fig. 2. Example KA for escalating to root privilege from a local login session. Choose an existing login session on that host, and then try to escalate the privilege of that session to root privilege.

and components. KAs are associated with particular goals, the purpose for which they are to be executed. One may also specify the *context* that limits the conditions under which a KA may be executed. Figure 2 shows a KA for achieving root privilege on some host. Note that it is only applicable when the agent is already logged into that host.

Within a KA definition, one may invoke a special procedure, *achieve_goal*(G, S), to achieve a subgoal. We give pseudo-code for *achieve_goal*(G, S) in Figure 3. *achieve_goal*(G, S) works as follows: first, check to see if the goal already holds. If so, simply return success. If the goal does not hold, the agent must do something to make it hold. Check to see if there are any methods available. If not, then return failure. If there are methods, choose one and invoke it. If the chosen method has succeeded, we're done. Otherwise, decide whether or not to persist in achieving the goal, and either try again, or give up. In our model, the probability of persistence is a simple constant parameter, but it would be trivial to extend the model to make the persistence probability sensitive to the goal in question.

In general, a single KA may have multiple subgoals. For greater convenience, the KA will monitor all of its subgoals for failure. If a subgoal fails irrecoverably, then the parent KA will fail also (possibly triggering replanning). We do this because subgoals are meaningful only in the context of an over-arching plan. For example, one might have a plan to travel by buying a train ticket and then boarding the corresponding train. If the subgoal of acquiring a ticket fails completely, then the entire plan fails, and the agent will try some other means of achieving his goal, instead of boarding the train with no ticket.[5]

3.4 Future enhancements

There are a number of issues in attacker modeling that we did not handle in this first version of our simulator. The first is that we allow only a single attacker.

[5] Assuming it is impossible to purchase a ticket on the train.

```
proc achieve_goal(G, S)
  begin
    if G
      then S := true;
    elsif all_methods(G) = ∅
          then S := false;
           else
                π ms, m
                ms := all_methods(G);
                m := choice(ms);
                call m;
                if G
                  then S := true;
                elsif stochastic_persist(G)
                      then achieve_goal(G, S),
                       else S := false;
                fi
    fi
  end
```

Fig. 3. Pseudo-code definition of *achieve_goal*. G is an "in" parameter, the goal expression. S is an "out" parameter, a Boolean value indicating success or failure in achieving the goal.

This is not as great a limitation as it sounds, since we *do* allow multiple sites of attack. So our current framework is adequate to model multiple, coordinated attackers. However, for future applications, multiple attackers should be added. Such models are necessary to allow researchers to experiment with techniques for attributing attacks to multiple different attackers and with disentangling the concurrent activities of truly threatening attackers from the background activities of "ankle-biters."

We also skirted the issue of the *duration* of actions. The concurrency semantics we use is strictly a matter of interleaving and no attempt is made to distinguish between actions that consume different amounts of time. There is already a treatment of actions with variable durations for the situation calculus, together with a corresponding Golog interpreter [23, Chapter 7]. Therefore, adding temporally-extended actions will not add any theoretical difficulty. However, it will add substantial complication to the modeling and knowledge-engineering process for the simulator. We simply do not have good estimates of the durations of the various actions and processes, and these durations bear complex relations to the state of the environment (e.g., network and host load). Nevertheless, if we are to incorporate features like (authorized) background traffic models, we will have to address this issue.

The current system has only deterministic actions. For example, the exploits will always work if and only if they are executed against a vulnerable target. Note that even though the actions are deterministic, *the simulated attacker is*

not. The attacker's choice of plans, exploits and targets, and its reaction to failures are all stochastic phenomena, lending a great deal of variance to even simple scenarios.

Even so, limiting ourselves to deterministic actions is obviously a substantial oversimplification. Therefore we must incorporate actions that succeed only with a certain probability. The formal framework for stochastic actions already exists [2]. As with action durations, the primary difficulty is the challenge of knowledge acquisition.

Most of the systems for working with the situation calculus and golog assume a finite domain of entities. The finite domain is necessary for the practical inference methods to succeed (cf. [23, Chapter 9], [8]). This limitation is not at all appropriate to modeling computer intrusion situations. Modeling interactions with software entities require us to be able to model the construction and (to a lesser extent) destruction of entities. For example, if we wish to talk about login sessions, we must be able to talk about their creation by the act of logging in to a host. One solution would be to create *ab initio* a fixed pool of *potential* sessions, and talk about them being actually there or not [18]. This solution poses two problems. First, it commits us to a fixed pool, and our simulator will fail if we guess wrong and overflow the preallocated pool. Second, it means that we will often find ourselves quantifying over unnecessarily large domains. For the moment, we have programmed ourselves out of this problem by restricting the cases where open domain entities can appear and by making sure we do not make inappropriate queries about those entities. This is an unsatisfactory state of affairs, however, and calls for some further technical work. There has been some preliminary work in the area of "softbots" [6], on exploiting local closed world knowledge [7]. This is related, but limited to trying to provide intelligent agents enough knowledge about action effects to predict whether they can act successfully, not to handle projection for simulation.

The current system has a patchy treatment of agents' knowledge and beliefs. We have put in special-purpose fluents to track the attacker's knowledge of key facts like the identity of hosts and user passwords. However, we have not extended this to a full knowledge model for the attackers. As with stochastic actions and actions with variable durations, there is already some available theory for such a model [25], [23, Chapter 11]. However, the current solutions are not acceptable for our purposes. The problem is that the existing solutions assume that agents will differ only in their knowledge of the *state* of the world. They must agree on the *physics* of the world — particularly on how all the actions will affect the state of the world. This is simply not appropriate for simulating computer attackers. Typical attackers — especially the "script kiddies" — will shape their attacks to the particular tools that they have. Furthermore, one can often determine useful things about an attacker based on ill-chosen actions. For example, one sometimes sees a script kiddie gain access to a Unix host, and attempt to execute Windows commands there. This is an area for further research.

A final area for work is less theoretically interesting, but perhaps the most important. This is the creation of better tools to work with our attacker simulation

tool. There are two most critical needs. The first is for some compile-time and run-time validation of model components. The indigolog interpreter is written in Prolog. This is desirable for many reasons, most notably because it permits a direct implementation of the semantics of the language. On the other hand, it means that we have a very crude compilation environment with very poor detection of ill-formed actions or tests. The system badly needs some form of type-checking or other verification. Currently all of these errors must be detected by the programmer using the debugger. This debugger should also be improved. Currently, one must use the Prolog debugger, which plunges the programmer into the details of the interpreter's implementation. It would be better to build a special-purpose debugger for the simulator application, that would focus on the golog semantics and suppress the details of their Prolog implementation. Finally, we have made a number of extensions to the syntax of the language to make it easier to write. However, further work could certainly be done in this area.

4 Modeling the Cyber Attack Domain

In this section, we discuss how we model cyber attack domains using our tool. After that, we present a specific example, the "Frostbite Falls" scenario. We then describe a simulated attack, generated by our simulator. This attack will show how the simulated attacker can use multiple alternative plans and exploits, to achieve its end. We will see that the attacker adapts to the outcomes, both success and failure, of earlier actions.

Our modeling of cyber attack domains has been somewhat ad hoc, but our modeling has a clear rationale, and our modeling decisions should be understood in the light of this rationale. The first component to this rationale is that cyber attack modeling is being done in order to support more cost-effective intrusion research. The second component is that we are interested in studying the way intelligent systems can combine reports from multiple sensors into an overall situation assessment.

The purpose of the cyber attack modeling tool is to allow more cost-effective intrusion research, with repeatable tests. We are particularly interested here in extended attack scenarios: not just the isolated deployment of a given exploit. There are three obstacles we'd like to overcome. The first is the cost of testing an intrusion scenario. Doing so involves building a test network, and then restoring the network's state after any destructive modifications by the attacker. The second obstacle is the requirement to have expert humans involved in performing the intrusions we would like to study. The final obstacle is the sheer time cost of exploring multiple variations of a single attack plan. It's common for red teams to develop attack trees that contain many ways of attacking a particular network. But it is very difficult to explore these at all thoroughly because of the time needed and the requirement for direct human involvement.

We are not interested in the detailed modeling of individual events. Instead, we are interested in the way a number of different sensors will report on the

same events. We are particularly interested in how those reports can be fused together in ways that exploit background knowledge.

Our purposes in this project, then, dictate a relatively abstract level of modeling. For example, we have avoided modeling the details of network traffic and network protocols. We have also avoided modeling the details of the file systems of the computers that are attacked. The primary consideration in this abstraction has been to develop a simulation that can be run far more quickly than a true version of the attacks. Compare this to network simulations intended to assess the performance of various network protocols, which typically run at only some fraction of the protocol itself. Further, since we are interested in what attacks will look like through the eyes of sensors, rather than experimenting with sensor designs, we avoid modeling the exact phenomena that will cause sensors to trigger. Of necessity, this will cause some inaccuracies, but we feel the price we pay is worthwhile, at least for this application.

4.1 The Frostbite Falls Scenario

The Frostbite Falls scenario concerns the attack made by a cracker, whom we'll call b0r15, on a network that contains an Oracle database (see Figure 4). b0r15 wants to gain access to the Oracle database (on the host `fellini`) order to corrupt its contents. However, initially b0r15 only knows the IP address of Frostbite Falls' DNS server, he does not know that `fellini` is the target. Further, b0r15 does not know any exploit that will directly allow him access to `fellini`; he will have to gain access indirectly, through another host on the defended network. This means that the attack will go through all of the classic cycles of reconnaissance, initial foothold, and exploitation and consolidation.

4.2 Attacking Frostbite Falls

The easiest way to use the cyber attack modeling tool is to experiment to with our Frostbite Falls scenario model and the attached KAs and primitive actions. Table 3 shows our hacker, b0r15 trying to gain access to the Frostbite Falls Oracle database in order to insert phony orders.

The transcript in Table 3 was generated by a top-level plan that has three steps:

1. prepare to attack (this is primarily concerned with getting b0r15 logged into a workstation he owns so that he can attack);
2. conduct reconnaissance;
3. get access to the oracle database, wherever it is.

The Golog code was as follows:

begin
$(\pi sess)start\_work(sess)$;
$b0ri5\_recon$;
$(\pi oh)?(known\_service(oh, oracle);\ achieve\_goal(access(oracle, oh)))$;
end

Table 3. Sample transcript from the Frostbite Falls scenario. Lines preceded by arrows indicate goal achievement.

```
1   login(boris,b0ri5,bpass,_session0)
2   =======>logged_into(boris)
3   zone_transfer(besson,boris)
4   ping_sweep(boris,ip(192,168,2,*))
5   ping_sweep(boris,ip(192,168,3,*))
6   ping_sweep(boris,ip(192,168,1,*))
7   port_sweep(boris,bergman)
8   port_sweep(boris,besson)
9   port_sweep(boris,fellini)
10  port_sweep(boris,kubrick)
11  port_sweep(boris,landis)
12  port_sweep(boris,lucas)
13  rlogin(boris,kubrick,rocky,_session1)
14  rlogin(boris,kubrick,rocky,_session2)
15  neptune(boris,lucas)
16  =======>neg(tcp_available(lucas))
17  session_hijack_add_perm_all(rocky,kubrick,lucas)
18  rlogin(boris,kubrick,rocky,_session3)
19  =======>logged_into(kubrick)
20  ftp(dtappgather)
21  =======>available(dtappgather)
22  dtappgather(_session3)
23  dtappgather(_session3)
24  email(sadmindex)
25  =======>available(sadmindex)
26  sadmindex(_session3)
27  =======>root_privileged(_session3)
28  =======>root_privileged_on(kubrick)
29  magic_transfer(sniffer)
30  =======>available(sniffer)
31  install_sniffer(_session3,kubrick)
32  =======>access(oracle,fellini)
33
34  yes
```

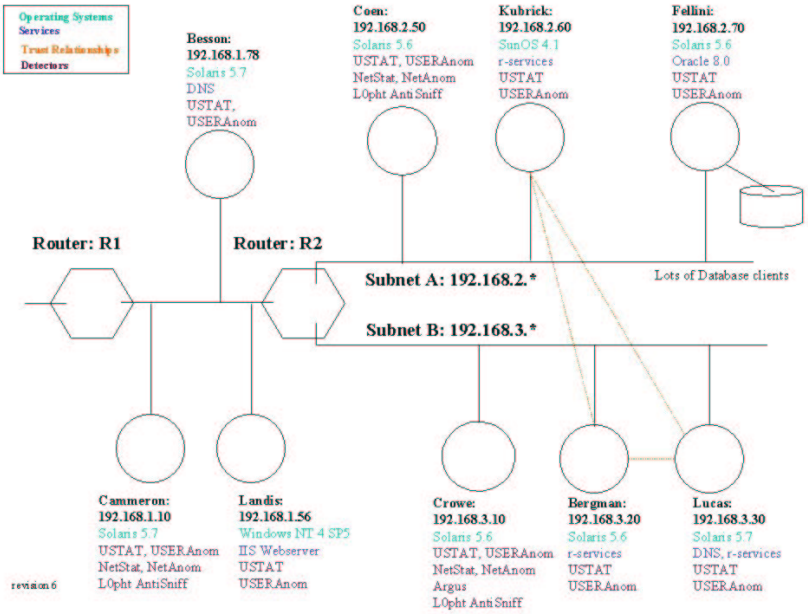

Fig. 4. Frostbite Falls network topology.

In Table 3 we see b0r15 first login to his own workstation (lines 1-2). Then he conducts his reconnaissance (3-12). He first tries a zone transfer from the DNS server of the network, then does an IP sweep. Finally, he does portsweeps to see what services are being run on the individual hosts. b0r15 is not stealthy!

b0r15 has a number of possible means of attack on the Oracle host. In this transcript he has chosen to attack by attempting to sniff Oracle passwords out of network traffic. This plan requires him first to achieve root privilege on some other host on the network. With root privilege, he can install the sniffer and then log into the oracle host.

b0r15 chooses `kubrick` as his stepping-stone on the way to attacking `fellini`. Recall that his plan calls for him to gain root access to `kubrick` as a means to install a sniffer. To gain root access, b0r15 chooses to gain local user level access to `kubrick` and then escalate his privilege (he could also have tried a more direct remote-to-root attack). b0r15 first tries a simple rlogin to `kubrick`, on the off-chance that the rlogin services have been left unsecured (13). Note that b0r15 tries this twice (13-14); the simulator provides for the possibility of persistence.

After two tries, b0r15 gives up on trying to get through an unguarded rlogin, and adopts session hijacking as an alternative. `kubrick` trusts `lucas` for the purposes of rlogin. b0r15 chooses the "neptune" denial of service attack from the set of alternatives, and brings down `lucas`'s TCP stack (15-16), allowing him to impersonate a legitimate user, and set up an rhosts file on `kubrick` (17).

Now that he has user-level access, b0r15 moves on to the goal of being `root_privileged_on kubrick`. He first tries the `dtappgather` exploit, mistakenly, since `kubrick` is not vulnerable to it (20-23). Note that in order to try this exploit he first transfers the the exploit code to `kubrick` via ftp (20). b0r15 gives up on `dtappgather` after trying it twice, and seizes on the `sadmindex` exploit. This time, instead of ftping the exploit, he decides to use email (24) to achieve the goal of `available(sadmindex)` (25).

With root-privilege secured, b0r15 transfers a sniffer onto `kubrick` (29-30). He does this using "`magic_transfer` (29), an action which is meant to stand in for any kind of covert channel the defenders have not yet seen. Thus, to the defenders this channel is, effectively "magic." Note that the addition of "magic" actions like this allow us to experiment with situations where we confront attackers with exploits not previously known to us. With the sniffer installed, b0r15 can get the password to Oracle user accounts, so he has achieved his goal of `access(oracle,fellini)`, and is done (32).

5 Related Work

We have already mentioned some of the most directly relevant work on attack modeling. Both Cuppens and Ortalo [4] and Templeton and Levitt [29] have developed modeling langauges based on actions with preconditions and postconditions. One of our contributions is to marry precondition/postcondition intrusion modeling with the situation calculus, which provides a rich and sound semantics.

At least two groups have used model-checking techniques and attack models to assess network vulnerabilities [24, 28]. This work is similar to ours in projecting the effects of action sequences. There are two important differences. First, the syntax and semantics provided by the model checkers' temporal logics are less expressively powerful and modular than the situation calculus. Second, these researchers are not interested in *naturalistic* modeling of computer attackers. Their interest lies in identifying network vulnerabilities, for which it is sufficient to consider the worst case attack, computed by the model-checkers' exploration of the attack space. We are interested in modeling the full situation, for which we must consider not only the most competent attacker, but the full range of phenomena. These two objectives are complementary, rather than conflicting, and it would be interesting to see whether the different efforts could benefit from each others' modeling efforts.

6 Conclusions

We have described a comprehensive approach to computer network attack simulation. Computer security researchers need such simulations in order to carry out large-scale, repeatable experiments in computer security. The situation calculus and the Golog situation calculus programming language, suitably extended, can provide a theoretical and practical basis for such simulations. We have provided a number of Golog extensions, most notably goal-directed procedure invocation,

to better model cyber attackers. Our prototype simulator can simulate a single attacker, who is able to synthesize full network attacks from a library of plans and primitive actions, reacting to successes and failures it encounters.

Acknowledgements

Thanks to Maurice Pagnucco for much assistance working with Indigolog and to Maurice, Hector Levesque, and the University of Toronto for providing the Indigolog interpreter. Thanks to Keith Golden for helpful comments based on his experience with softbot planning. Thanks to the Argus/Scyllarus team, and Dick Kemmerer and Giovanni Vigna for the Frostbite Falls scenario. This material is based upon work supported by DARPA/ITO and the Air Force Research Laboratory under Contract No. F30602-99-C-0017. The work described here was done while the author was employed at Honeywell Laboratories.

References

[1] American Association for Artificial Intelligence, *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, Menlo Park, CA, July 2000. AAAI Press/MIT Press.

[2] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun, "Decision-theoretic, High-level Agent Programming in the Situation Calculus,", in *Proceedings of the Seventeenth National Conference on Artificial Intelligence* [1], pp. 355–362.

[3] M. E. Bratman, "What is Intention?," in *Intentions in Communication*, P. Cohen, J. Morgan, and M. Pollack, editors, chapter 2, pp. 15–31, MIT Press, Cambridge, MA, 1990.

[4] F. Cuppens and R. Ortalo, "LAMBDA: A Language to Model a Database for Detection of Attacks," in *RAID*, H. Debar, L. Mé, and S. F. Wu, editors, volume 1907 of *Lecture Notes in Computer Science*, pp. 197–216. Springer, 2000.

[5] DARPA and the IEEE Computer Society, *DARPA Information Survivability Conference and Exposition(DISCEX-2001)*, 2001.

[6] O. Etzioni, "Intelligence without Robots: A Reply to Brooks," *AI Magazine*, vol. 14, no. 4, pp. 7–13, 1993.

[7] O. Etzioni, K. Golden, and D. Weld, "Tractable Closed World Reasoning with Updates," in *Principles of Knowledge Representation and Reasoning:Proceedings of the Fourth International Conference*, J. Doyle, E. Sandewall, and P. Torasso, editors, pp. 178–189. Morgan Kaufmann Publishers, Inc., 1994.

[8] A. Finzi, F. Pirri, and R. Reiter, "Open World Planning in the Situation Calculus,", in *Proceedings of the Seventeenth National Conference on Artificial Intelligence* [1], pp. 754–760.

[9] R. J. Firby, "An Investigation in Reactive Planning in Complex Domains," in *Proceedings of the Sixth National Conference on Artificial Intelligence*, pp. 196–201. AAAI, Morgan Kaufmann Publishers, Inc., 1987.

[10] C. W. Geib and R. P. Goldman, "Plan recognition in intrusion detection systems,", in *DARPA Information Survivability Conference and Exposition(DISCEX-2001)* [5], pp. 46–55.

[11] M. Georgeff and A. Lansky, "Procedural Knowledge," *Proceedings of the IEEE, Special Issue on Knowledge Representation*, vol. 74, pp. 1383–1398, October 1986.

[12] M. P. Georgeff and F. F. Ingrand, "Real-Time Reasoning: The Monitoring and Control of Spacecraft Systems," in *Proceedings of the Sixth Conference on Artificial Intelligence Application*, pp. 198–204, 1990.

[13] G. D. Giacomo, Y. Lesperance, and H. Levesque, "ConGolog, A concurrent programming language based on the situation calculus," *Artificial Intelligence*, vol. 121, no. 1-2, pp. 109–169, 2000.

[14] G. D. Giacomo, H. J. Levesque, and S. Sardiña, "Incremental execution of guarded theories," *ACM Transactions on Computational Logic*, vol. 2, no. 4, pp. 495–525, October 2001.

[15] R. P. Goldman, W. Heimerdinger, S. A. Harp, C. W. Geib, V. Thomas, and R. L. Carter, "Information Modeling for Intrusion Report Aggregation,", in *DARPA Information Survivability Conference and Exposition(DISCEX-2001)* [5], pp. 329–342.

[16] H. Grosskreutz and G. Lakemeyer, "On-Line Execution of `cc-Golog` Plans," in *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pp. 12–18, Los Altos, CA, August 2001, Morgan Kaufmann Publishers, Inc.

[17] A. R. Haas, "The case for domain-specific frame axioms," in *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*. Morgan Kaufmann, 1987.

[18] Y. Lesperance, August 2001. Personal communication.

[19] H. J. Levesque and M. Pagnucco, "Legolog: Inexpensive Experiments in Cognitive Robotics," in *Proceedings of the Second International Cognitive Robotics Workshop*, Berlin, Germany, August 2000.

[20] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl, "GOLOG: A Logic Programming Language for Dynamic Domains," *Journal of Logic Programming*, vol. 31, no. 1-3, pp. 59–83, 1997.

[21] J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," in *Machine Intelligence*, B. Meltzer and D. Michie, editors, volume 4, Edinburgh University Press, Edinburgh, 1969.

[22] R. Reiter, "The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression," in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, Vladimir Lifschitz (Ed.), Academic Press*, 1991.

[23] R. Reiter, *Knowledge in Action*, MIT Press, Cambridge, MA, 2001.

[24] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in *Proceedings 2000 IEEE Computer Society Symposium on Security and Privacy*, pp. 156–165, May 2000.

[25] R. B. Scherl and H. J. Levesque, "The Frame Problem and Knowledge-producing Actions," in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 689–695, Menlo Park, CA, 1993, AAAI Press/MIT Press.

[26] B. Schneier, *Secrets & Lies*, John Wiley & Sons, 2000.

[27] L. Schubert, "Monotonic Solution of the Frame Problem in the situation calculus," in *Knowledge Representation and Defeasible Reasoning*, J. H.E. Kyburg, editor, pp. 23–67, Kluwer Academic Publishers, 1990.

[28] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated generation and analysis of attack graphs," in *2002 IEEE Symposium on Security and Privacy (SSP '02)*, pp. 273–284, Washington - Brussels - Tokyo, May 2002, IEEE.

[29] S. J. Templeton and K. Levitt, "A Requires/Provides Model for Computer Attacks," in *Proceedings of the New Security Paradigms Workshop*, sep 2000.