2002-06

Software Decoys: Intrusion Detection and Countermeasures

Michael, James Bret

Monterey, California. Naval Postgraduate School

Proceedings of the 3rd IEEE Workshop on Information Assurance, West Point, NY, June 2002.

James Bret Michael, *Senior Member, IEEE*, Mikhail Auguston, Neil C. Rowe, and Richard D. Riehle

Software Decoys: Intrusion Detection and Countermeasures

Abstract—We introduce the notion of an intelligent software decoy, and provide both an architecture and event-based language for automatic implementation of them. Our decoys detect and respond to patterns of suspicious behavior, and maintain a repository of rules for behavior patterns and decoying actions. As an example, we construct a model of system behavior from an initial list of event types and their attributes in the interaction between computer worms and an operating system. The model represents patterns of suspicious or malicious events that the software decoy should detect, and specific actions to be taken in response. Our approach explicitly treats both standard and nonstandard invocations of components, with the latter representing an attempt to circumvent the public interface of the component.[a]

***Index terms*—Behavior modeling, computer security, computer worm, event trace, software decoy, intrusion detection, intrusion tolerance**

This paper appeared in the Proceedings of the 3rd IEEE Workshop on Information Assurance, West Point, NY, June 2002.

I. INTRODUCTION

Software components with poorly designed interfaces are susceptible to misuse or modification by rogue programs. Consider, for example, the result of unleashing the Morris Internet worm (*vid*. Spafford [1]): Some integral components of the UNIX operating system permitted the worm to propagate itself over the Internet. What were some of shortcomings of some versions of Unix in 1988? One was that some components permitted incorrect argument types to be executed, such as the *sendmail* program accepting commands instead of user addresses. Another weakness was that some components allowed for erroneous argument values to be passed to functions, such as a string of 536 bytes passed to the input buffer of the *fingerd* program: That string exceeds the size of the buffer, but *fingerd* and the functions it calls in the C language I/O library do not check, resulting in a buffer overflow.

Software patches were made to *sendmail* and *fingerd* in response to the Morris worm. The patches consisted of exception-handling routines for catching errors in input arguments to these programs. The effectiveness of the patches was low at first because of delays between the release and installation.

History continues to repeat itself. Distributed systems are plagued by worms descended from the Morris worm, such as the well-publicized "Code Red" worm [2] that exploits weaknesses in the public interfaces of components of the Microsoft Windows NT operating system that manipulate the registry.[1] The "patch-and-pray" approach to dealing with such weaknesses in interfaces has not been effective. Shortly after the introduction of the original, a variant of the Code Red worm appeared which took advantage of other weaknesses in the interfaces of those same components of the operating system [3].

In this paper we explore a new way of thinking about the challenges associated with protecting components within distributed systems from the effects of attacks. In particular, we develop the mechanisms needed to realize a modified version of what Michael and Riehle call an *intelligent software decoy* [4]. Intelligent software decoys adapt in order to tolerate both intrusions into systems by unauthorized users and misuse of components, rather than either indicating to the intruder or offending process that a violation of security policy has been detected or terminating the interaction with the process. Each software component has a *contract* consisting of a class invariant and one or more preconditions and postconditions; communication with a component is only permitted via its contract interface. Class invariants govern the nature and extent of any change to objects within a component by a nonstandard invocation (i.e., circumvention) of the component's public interface.

When a calling process passes arguments, via remote procedure call or remote method invocation, that violate the contract of a component, the component transitions from its nominal operating mode to a deception mode, in which it attempts to both deceive the calling process into concluding that its violation of the contract has been successful and assess the nature of the violation. For example, suppose a precondition in a contract asserts that input strings shall not exceed 512 bytes. If the component receives a string that violates this condition, then the component tries to continue to engage the calling process that such strings are acceptable while both protecting the component from the effects of the invocations and assessing the nature of the violation.

Here we describe an architecture and event-based language for automatic implementation of software decoys.

II. APPROACH

We start with the introduction of a precise *behavior model* for the system under consideration. This model is specified in terms of events and two binary relations over those events: precedence and inclusion.

We provide a formalism to specify rules for runtime intrusion detection and corresponding countermeasures based on behavior patterns over event traces and a catalog of decoy actions, such as blocking or substituting certain system commands. This implies an implementation based on automatic instrumentation for event detection derived from the behavior model. Intrusion detection rules are textually separated from the source code of the system, which allows for accumulating and formalizing knowledge of typical intrusion patterns and decoy strategies.

A. Event Traces

An *event* is an abstraction of any detectable action performed at runtime. An event has a beginning, end, duration, and some other attributes, such as program states at the beginning and end of the event, source code associated with the event, and so on. Two binary relations are defined for events. One event may precede another event. For example, one statement execution may precede another. In addition, one event may be included in another. For instance, a statement-execution event may appear inside a procedure-call or method-invocation event. Each of these binary relations defines a partial ordering of events. System execution may be represented as a set of events with the two basic relations between them—an *event trace*. An *event grammar* [5] is a set of axioms that determines possible configurations of events of different types within the event trace. We use the notion of computations over event traces to specify behavior patterns and decoy actions. This mechanism is a basis for automatic instrumentation of the source code.

B. Specification of Events

The behavioral model consists of the definition of event types and the attributes of the events. Our event grammar is used to describe the structure of events. The behavioral model consists of two parts: (i) a specification of *axioms* that define constraints on the behavior of components; this specification is given in terms of inclusion and precedence relations, and (ii) a description of *patterns of behavior* expressed in terms of event patterns.

The event grammar is not intended for actual parsing of an event trace. Each event defined in the behavior model should be detectable by some independent means, for example, by proper instrumentation of the source code. There always is a main event `execute-program`, which encompasses all other events. Event type and event attribute declarations, and event grammar rules (i.e., axioms) constitute the behavior model, or "lightweight" semantics specification for the system under consideration. The following is an example of an event-grammar rule that specifies that an event of type `execute-assignment` always contain (i.e., the inclusion relation) two ordered events: `evaluate-right-hand-part` and `perform-destination`.

```
execute-assignment::
    (evaluate-right-hand-part
     perform-destination)
```

Intrusion detection and other monitoring activities are defined in terms of this behavior model. This model opens the way for automatic source program instrumentation, and provides formalism for describing different kinds of behavior patterns, for example, typical intrusion patterns.

C. Computations over Event Traces

Event pattern A matches successfully any event of the type *A*. The event pattern can contain additional context conditions, which typically involve event attributes, for instance, `X: perform-destination & source-code(X) = 'V'`. This event pattern matches an event of type perform-destination, such that source-code attribute of this event is 'V'. Notice the use of auxiliary variable `X` to denote the event under consideration.

Event patterns are used in *aggregate operations* to select sequences of events from other event sets. For example, `[X:utility-call & name(X) = 'sendmail' FROM execute-program]` yields a sequence of events matching the pattern `X: utility-call & name(X) = 'sendmail'` selected from the whole event trace. An aggregate operation may produce a result different from just a sequence of events. For instance, SUM/`[X:utility-call & name(X) = 'sendmail' FROM` `execute-program APPLY duration(X)]` yields a total duration of events selected.

Event patterns can be combined in order to specify *path expressions*: patterns of event sequences. For instance, the following path expression specifies a sequence of events starting with event of type *A* followed by zero or more events of types *B* or *C*: `A (B | C)*`

A *probe* is a Boolean expression involving event attributes and standard arithmetic and logic operations. It might also contain calls of subroutines written in some general programming language. Probes may be included in path expressions interlaced with event patterns. A probe is evaluated immediately after the preceding event pattern has been matched successfully. A probe is successful if it evaluates to the value True. For example, the following path expression specifies a utility call that does not change the value of variable V.

```
X: utility-call
     Probe
        (value(V at begin X) =
         value (V at end X))
```

These operations provide a basis for computational tasks such as filtering events to create views of the event trace subspace, evaluating the truth of assertions, and computing specific values (e.g., counting the number of events in a sequence or the maximum elapsed time for an event pattern). An important feature of this formalism is that computations over event traces are separated from the source code itself, and the instrumentation of the source code can be automated, based on the behavior model (i.e., event grammar) and the text of the trace computation.

D. Code Instrumentation

Code instrumentation is needed to recognize patterns of events and to perform computations over the event trace. Our approach is to selectively specify and automatically instrument, on a component-by-component basis, only those event types and event attributes that are of interest. For example, one would likely specify events corresponding to popping (i.e., deleting an item from) the stack, reading, writing, or handling threads (e.g., create, reschedule, delete) as events of interest in the context of the behavior of components comprising an operating system; only the source code corresponding to these and other events of interest would be instrumented. For other components within the operating system, there may not be any events of interest and therefore the source code of those components would not be instrumented.

An automated tool for generating the instrumentation would accept as input the program's source code and the text of trace computation. We define the event pattern language and aggregate operations in such a way that event detection, event pattern matching, and other computations over event traces can be performed at runtime. As reported by Sekar and Uppuluri [6], similar instrumentation may result in approximately four percent runtime overhead. This effect of event-trace computations is important for high-performance systems, especially for components that are called frequently, such as those comprising either the scheduler or memory manager of an operating system.

As an example of how instrumentation would be selectively applied, consider a component that implements the Transmission Control Protocol (TCP); TCP is a frequently called component and known to be susceptible to certain types of attacks, such as that of the "Cheese" worm [7] that attempts to execute commands on a specific TCP port of a system and then masquerade as the *httpd* program on that system. The change to the *httpd* program via a Trojan horse is an example of an attack for which a class invariant could be used to guard against unauthorized changes to the behavior of this component. One of the events of interest would be the invocation of shell commands on a TCP port. Due to the fact that there is a sequence of calls resulting from the shell commands, the events at other components may need to be instrumented; this is the case for the Cheese worm. Some of the events of interest would be changing process names, overwriting system files (e.g., /etc/inetd.conf), and repeated attempts to restart a system command (e.g., *inetd*). The compiler used for generating instrumentation would only do so for these types of events. Specific examples of how to specify events, their attributes, and actions (i.e., responses to events) are given in the remainder of the paper for the Morris worm.

III. DOMAIN MODELS OF SYSTEM BEHAVIOR

Domain models of system behavior serve as the foundation from which to specify both patterns of suspicious behavior (i.e., slices of event histories) and the corresponding responses (i.e., actions) of software decoys. A domain model consists of a list of generic event types, along with their attributes, both of which are associated with a particular class of behavior and detectable. In this section, we report our work to date to develop a domain model of detectable events that can be triggered within an operating system by computer worms: that is, rogue programs that propagate themselves across computer networks. In other words, we treat the interaction between a worm and a particular type of software component as a unique domain for modeling purposes. Our motivation here is to both describe an event-trace language for implementing software decoys and give specific examples of how one would instrument code for runtime checking. Our examples are based on the behaviors of components of the Unix operating system with the Morris Internet worm. More recent worms, such as "Code Red" and "Cheese," are variants of the Morris worm in at least two ways: (i) they are composed from the same set of generic event types, or "building blocks," and (ii) share the same basic structure, in some cases initiating similar patterns of events.

A. Interaction between the Worm and Operating System

Although it is still debated as to whether the intent of the author of the worm was to just experiment out of academic interest and the worm "got loose" due to an egregious use of contracts at the interfaces of the components it called, the interaction of the worm with software components resulted in a temporary denial of service across much of the Internet.

The Morris worm interacts with the components of the operating system by repeatedly accessing a data structure (e.g., the system password and network configuration files), calling a program to obtain state information (e.g., *netstat*), or repeatedly invoking a program and permuting the arguments before each invocation (e.g., *rsh*). The worm also invokes sequences of functions and procedures, such as issuing the *DEBUG* command to the *sendmail* program followed by a sequence of commands (i.e., the vector portion of the worm program) to be acted on by procedures available in that operating mode.

In addition to accessing components, the Morris worm attempts to write to files (e.g., set a flag) and execute copies of itself in the form of Trojan horses. It also attempts to propagate itself while simultaneously eluding detection or spoofing by forking itself after a pre-specified number of tries at infecting other computers or elapse of time, and then deleting both the parent process and all of the temporary files created by the parent process.

B. Formal Description of the Domain Model

Here we provide a first approximation of the description of the model. We begin by specifying five event types and their attributes:

```
fingerd_call
     Attributes:    caller_id
                    begin_time
param_pass
     Attributes:    length
read
utility_call
     Attributes:    begin_time
                   caller_id
                   caller_process_type
sendmail_call
     Attributes:    in_debug_mode
```

Next we introduce an axiom that specifies the event `fingerd_call` contains another event, `param_pass`:

Axiom: `fingerd_call:: ( param_pass)`

Based on the foregoing specification of events and axioms, we can start to build a behavioral specification for the Morris worm as follows:

```
fingerd_call:: ( x: param_pass )
    & length(x) > max_buffer_size
    read +
    probe( buffer_overflow)
```

This behavior pattern, described in terms of a violation of the contract for *fingerd* calls, states that first an event of the type `fingerd_call` is detected, such that the value of attribute `length` of the included event `param_pass` exceeds a given constant `max_buffer_size`; in this case, `x` is a tentative name associated with the event `param_pass`. Next, one or more events of the type `read` should appear, and finally a probe (i.e., an evaluation of a Boolean expression) specifying buffer overflow should evaluate to the value True. Successful detection of those events and conditions in the specified order indicates that the system might be under attack by the Morris worm or some other worm. Now let us specify yet another behavior pattern:

```
x: sendmail_call::
    ( [ utility_call + ]
    && CONST(caller_id)
    && CONST(caller_process_type)
    && FREQUENCY(time_interval)
    )
    & in_debug_mode(x) = True
```

Here `sendmail_call` is specified with the attribute `in_debug_mode` equal to the value True. This event contains a sequence of `utility_call` events with the same `caller_id` and `caller_process_type` attribute values, such that the time between those `utility_call` events does not exceed `time_interval`.

C. Summary

In summary, our event-trace language integrates the language constructs that are necessary for specifying event patterns, probes, and actions. An event pattern is a particular interval-based signature constructed from a structural model of events of interest (e.g., system calls to write to a file). The probe is a Boolean expression; for instance, a probe can be used to compare the values of two attributes over the same interval of time, or the value of the same attribute at different points in time. An action is a message that is generated based on the recognition that a particular pattern of behavior has been observed: The pattern of behavior can be composed from one or more slices of the same or different event traces. The messages are used to trigger rules that embody the target component's decoy-mode response to its interaction with the calling process whose interaction with the component may be malicious or benign.).

IV. DECOY STRATEGY

Decoy methods can show a wide spectrum of complexity. A key dimension is information in the mathematical sense they provide to the attacker [8]. A decoy that provides the same canned response to a user command in all circumstances transmits zero bits of information. A decoy that simulates an arbitrary protected file by generating a fake file with random choices transmits information equal to the logarithm of the number of possible fake files it can create for a given request. A decoy that simulates a denial-of-service attack, providing additional operating-system delays monotonically increasing with the number of requests in a recent time interval, transmits information equal to the logarithm of the number of distinguishable time intervals it can provide. A decoy that simulates attempts to change a module of the operating system transmits information equal to the logarithm of the size of that module (or perhaps the part of it relevant to a particular attacker activity). It might seem that the ultimate decoy would simulate all aspects of a computer and its software as proposed by Thimbley *et al.* [9], requiring information equal to the logarithm of the size of the complete description of the computer and programming environment, and thus be equivalent to a universal Turing machine. However, we claim a truly ultimate decoy would be "intelligent" and need to do more: It would need to analyze properties of arbitrary code of viruses and Trojan horses to predict their conceptual consequences. But such ultimate decoying appears impossible in light of the undecidability of nontrivial properties of recursive functions (*vid.* Rice's Theorem [10]).

Applying these ideas to the Morris worm, the propagation of the worm could be decoyed by having the operating system count the number of recent buffer-overflowing *fingerd* calls (requiring a single integer of memory) and provide additional process-suspension time in all operating-system activities; this could most easily be done by just inserting delays into a frequently used utility like file reading, delays that would only occur for the process threads created by the perceived attacker and would be monotonic with the number of ongoing threads they have created. The *sendmail* trapdoor could be decoyed by permitting normal debugging behavior except for commands writing or executing parts of memory, which would appear to execute but would be prevented from changing memory.

Consider some examples of decoy strategies for the Morris worm. The propagation of the worm could be decoyed by providing additional process-suspension time for all utility calls within the same process, creating an illusion to the intruder that a delay has been introduced (assuming that the intruder monitors response times) while the other non-offending processes will continue to execute unhindered.

```
detect [ x: fingerd_call::
    ( y: param_pass )
```

```
        & length(y) > max_buffer_size
        read + ] && CONST(process_id)
        probe (buffer_overflow)
    from execute-program
    do enable delay (t) to z: utility_call
        & process_id(x) = process_id(z)
```

The *sendmail* trapdoor could be decoyed by directing its invoker to a decoy function `protected_read()` that simulates the behavior with correct read commands and by deactivating write commands via the decoy function `dummy()` to substituting write operations:

```
    detect x: sendmail_call::
        ( [ utility_call + ]
        && CONST(caller_id)
        && CONST(caller_process_type)
        && FREQUENCY(time_interval)
        ) & in_debug_mode(x) = True
    from execute-program
    do substitute z: write
        & process_id(x) = process_id(z)
        by dummy()
        substitue w: read
        & process_id(x) = process_id(w)
        by protected_read()
```

Finally, the *fingerd* buffer-overflow to change the operating-system stack could be simulated via the instrumentation of the pop operations on the stack to simulate a jump to the new designated area of memory and execution of its commands with the current execution environment, requiring a full simulation of what happens there but on a copy of the operating system designated only for the offending process. Decoy methods like these can be generalized for broader applicability (by inserting code to call a general decoy monitor into many executables and parts of the operating system) or specialized for efficiency (by compiling separate decoy-enabled versions of executables that can be called when suspicious activities occur) depending on design priorities.

A software decoy should try to fool an intruder into thinking it is real. A necessary condition is that it returns most of the bits of information that would be returned by the real component corresponding to the simulated component. (This is not a sufficient condition because the type of information returned may be important even for the same number of bits.) A necessary information-theoretic condition for this is that the size of the decoy in bits must be most of the size of the amount of information it must transmit—and in the case of complex decoys that are too hard to summarize in an input-response table, perhaps larger still to model the decoying behavior in an easy-to-debug and easy-to-update way. However, in many practical situations it is sufficient that the decoy provide only a sample of all possible behaviors in response to only a sample of possible attacker inputs. For instance, a software decoy need only provide convincing fakes for the protected files that an attacker asks to see. For such situations, a good measure of decoy quality is the difference between the number of bits expected by the attacker and the number of bits transmitted. So if the same response is given to each of three actions, the number of bits transmitted remains the same as if one response is give to one action, but the number of bits expected is tripled.

V. SOFTWARE DECOY ARCHITECTURE

We turn now to the implementation of the event-trace language to support software decoys. The structural (i.e., domain behavior) model will consist of event types, event attributes, and axioms. Event types and attributes should be detectable independently.

A specialized compiler will selectively instrument software components for the purpose of detection of events and event attributes. It might be necessary from a performance perspective to optimize the instrumentation. In addition, the compiler may need to be constructed with an option that permits the user to store event traces; for instance, the user may want to use these traces for debugging purposes.

The runtime architecture will include mechanisms for monitoring behavior-pattern implementation, via the use of state transition diagrams and buffers to maintain event-attribute values.

Fig. 1 represents one possible architectural solution for implementing intelligent software decoys, in particular, for software components within an operating system. The separation of the repository of rules and behavior patterns from the supervisor supports the maintenance of the repository without changing the implementation (i.e., component wrappers and supervisor); the repository and behavior model are confidential within the system under protection.

Execution of system commands contributes the main part of the event types in our behavior model. Each system command participating in the model is enclosed in a wrapper, which is responsible for sending to the supervisor messages about event beginning and end and event attributes. The wrapper also contains a lightweight framework for implementing decoy actions, e.g. substituting the execution of the command by another subroutine execution, delay implementation, etc. Behavior pattern detection is based on state transition diagrams maintained in the supervisor. Interpreter within the monitoring kernel is responsible for rule interpretation and triggering decoying actions.

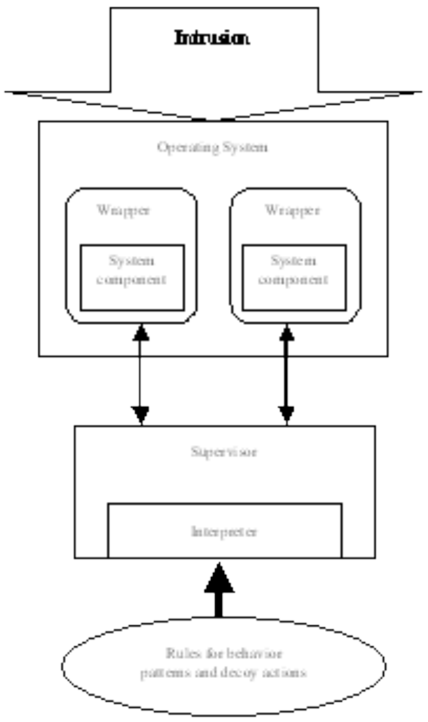

Fig. 1. Software decoy architecture.

VI. RELATED WORK

The notions of event, path expression, and assertion language are well known in testing and debugging automation research. An event-based debugger for the C programming language called Dalek [11] provides user defined events that typically are points within a program execution trace. A target program has to be instrumented manually in order to collect values of event attributes. The path expression technique was introduced for specifying parallel programs in [12]. This technique has been used in several projects on high-level debugging tools, (e.g. in [13]), where path rules are suggested as a kind of debugger command. Assertions (or annotations) currently in use are mostly based on Boolean expressions attached to selected points of the target program (e.g., the *assert* macro in C). The ANNA [14] annotation language for Ada supports assertions on variable and type declarations. In the TSL [15] annotation language for Ada, the notion of event is introduced to describe the behavior of tasks. The RAPIDE project [16] provides a rich event-based assertion language for software architecture description. In [17], a practical approach to assertions for the C language is advocated. Our language for computations over traces provides a flexible means for writing both local and global assertions, including those about temporal relations between events. It supports all kinds of assertions used in the systems above in a uniform framework.

Sekar and Uppuluri [6] describe a high-level formalism for specifying intended program behaviors using patterns over sequences of system calls. The patterns also capture conditions on the values of system-call arguments. Their intrusion prevention and detection system is based on intercepting system calls, comparing them against the specifications, and disallowing calls that deviate from the specifications of valid behaviors. In their approach, it is possible to modify a system call before it is delivered to the operating system's kernel, permitting the system to react before the execution by a process of any damage-causing system call. The paper also presents a low-overhead algorithm for matching runtime behaviors against specifications. Based on a series of experiments, Sekar and Uppuluri report that the overhead for implementing their monitoring algorithm on average is approximately four percent of the runtime.

We suggest a more powerful and expressive pattern specification language based on precise behavior models (event grammars) that will be able to capture a broader class of behaviors than those described by Sekar and Uppuluri. We also expect that our implementation strategy will be close to the guidelines presented by Sekar and Uppuluri, thus keeping the runtime overhead at or below four percent.

Neumann and Porras describe the architecture for EMERALD—an environment for anomaly and misuse detection—and subsequent analysis of the behavior of the systems and networks [18]. The architecture of EMERALD is based on component principles and can be adjusted to various platforms and configurations. A production-based expert system is used to analyze behavior patterns and signatures.

The use of event traces for reasoning about the behavior of information systems is not new. Event traces have been used to reason about the correctness of the execution of hardware-level instructions. For instance, Bressoud and Schneider [19] present an approach to fault tolerance based on the use of hypervisors that emulate the target hardware architecture: Each hypervisor serves as a virtual machine. The hypervisor is then instrumented to watch for sequences of instructions, within an epoch (i.e., interval of time), that are critical to the correct operation of replicated hardware within a system.

Working at a higher level of abstraction, Erlingsson and Schneider [20] introduce an approach known as Security Automata SFI Implementation (SASI). In this approach, the object code is instrumented so that as sequences of events occur, the trusted computing base, which in this case includes the software that performs object code analysis and object code modification, can use the event traces to reason about whether the traces are conformant to security policy (e.g., memory-protection policy).

Bressoud and Schneider [19] discuss some of the difficulties in instrumentation of virtual machines and reasoning over sequences of hardware instructions. Likewise, Erlingsson and Schneider admit that it is difficult to recognize patterns of interest from specifications of object and security policy represented in their security automaton language (SAL), and that the development of such specifications is an "awkward and error-prone" process. Their new approach is to rely on the use of "annotations of the object code that are easily checked and that expose application-level abstractions," rather than checking every machine-level instruction. Our formalism of decoys relies on specifying events for which the source code of components can be instrumented.

Sekar and Uppuluri [6] attempt to defend against buffer-overflow attacks by injecting code and addresses into memory, and transferring control flow to the injected code. They manually instrument system calls and their arguments for each component to be protected against buffer-overflow attacks. A run-time monitor then compares these patterns of nominal and anomalous behavior with that of the runtime behavior of the protected component. Our approach differs from that of Sekar and Uppuluri, especially in terms of the method of instrumentation: that is, in our approach, the software components are automatically instrumented.

Liu and Jajodia [21] introduce specialized decoying methods and strategies for transaction-level intrusion tolerance and damage confinement in database management systems (DBMS). The approach is bound to transaction events within the DBMS and relies on the use of filtering strategies. In contrast, our approach can be specialized to different types of events and countermeasures.

VII. CONCLUSION

Our approach explicitly treats the two types of invocations of components: standard and nonstandard, with the latter representing an attempt by a process to circumvent the public interface of the component. For standard types of invocations or components of legacy systems for which the source code is not available, then certain types of events of interest will not be detectable.

The behavior model based on event grammars provides a uniform framework for behavior-pattern recognition and decoying actions. In our opinion, this is a practical road to formalizing knowledge about typical intrusion patterns and constructing a flexible system of countermeasures.

VIII. ACKNOWLEDGEMENTS

This research is supported by the Naval Research Laboratory under contract no. N41756-01-WR-10433, the U.S. Army Research Laboratory under contract no. 40473-MA, and the National Research Council. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding

any copyright annotations thereon.

REFERENCES

[1] Spafford, E. H. The Internet worm: Crisis and aftermath. *Comm. ACM* 32, 6 (June 1989), 678-687.

[2] “Code Red” worm exploiting buffer overflow in IIS indexing service DLL. Incident Note IN-2001-08, CERT Coordination Center, Carnegie-Mellon Software Eng. Inst., Pittsburgh, Penn., July 19, 2001.

[3] “Code Red II:” Another worm exploiting buffer overflow in IIS indexing service DLL. Incident Note IN-2001-09, CERT Coordination Center, Carnegie-Mellon Software Eng. Inst., Pittsburgh, Penn., Aug. 6, 2001.

[4] Michael, J. B. and Riehle, R. D. Intelligent software decoys. In *Proc. Monterey Workshop: Eng. Automation for Software Intensive Syst. Integration*, Monterey, Calif.: Naval Postgraduate School (Monterey, Calif., June 2001), 178-187.

[5] Auguston, M. Tools for program dynamic analysis, testing, and debugging based on event grammars. In *Proc. Twelfth Int. Conf. Software Eng. and Knowledge Eng.*, Skokie, Ill.: Knowledge Syst. Inst. (Chicago, Ill., July 2000), 159-166.

[6] Sekar, R. and Uppuluri, P. Synthesizing fast intrusion detection/prevention systems from high-level specifications. In *Proc. USENIX Sec. Symp.*, Berkeley, Calif.: Usenix Ass. (Washington, D.C., Aug. 1999), 63-78.

[7] The “cheese” worm. Incident Note IN-2001-05, CERT Coordination Center, Carnegie-Mellon Software Eng. Inst., Pittsburgh, Penn., May 17, 2001.

[8] Evans, S., Bush, S. F., and Hershey, J. Information assurance through Kolmogorov complexity. In *Proc. DARPA Inf. Survivability Conf. and Exposition II*, vol. 2, IEEE (Anaheim, Calif., June 2001), 322-331.

[9] Thimbleby, H., Anderson, S., and Cairns, P. A framework for modelling Trojans and computer virus infection. *Comput. J.* 41, 7 (1998), 444-458.

[10] Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 89 (1953), 25-99.

[11] Olsson, R., Crawford, R., and Wilson, W. A dataflow approach to event-based debugging. *Software—Practice & Experience* 21, 2 (Feb. 1991), 19-31.

[12] Campbell, R. H. and Habermann, A. N. The specification of process synchronization by path expressions. In Goos, G. and Hartmanis, J., eds., *Proc. Int. Symp. Operating Syst., Lect. Notes Comput. Sci.*, vol. 16, Berlin: Springer-Verlag (Rocquencourt, Fr., Apr. 1974), 89-102.

[13] Bruegge, B., and Hibbard, P. Generalized path expressions: A high-level debugging mechanism. *J. Syst. Software* 3, 4 (1983), 265-276.

[14] Luckham, D., Sankar, S., and Takahashi, S. Two-dimensional pinpointing: Debugging with formal specifications. *IEEE Software* (Jan. 1991), 74-84.

[15] Luckham, D., Bryan, D., Mann, W., Meldal, S., and Helmbold, D. P. An introduction to task sequencing language, TSL version 1.5 (Preliminary version), Stanford Univ., Feb. 1, 1990, pp. 1-68.

[16] Luckham, D. and Vera, J. An event-based architecture definition language. *IEEE Trans. Software Eng.* 21, 9 (Sept. 1995), 717-734.

[17] Rosenblum, D. A practical approach to programming with assertions. *IEEE Trans. Software Eng.* 21, 1 (Jan. 1995), 19-31.

[18] Neumann, P. G. and Porras, P. A. Experience with EMERALD to date. In *Proc. First USENIX Workshop Intrusion Detection and Network Monitoring* (Santa Clara, Calif., Apr. 1999), 73-80.

[19] Bressoud, T. and Schneider, F. B. Hypervisor-based fault-tolerance. *ACM Trans. Comput. Syst.* 14, 1 (Feb. 1996), 80-107.

[20] Erlingsson, Ú. and Schneider, F. B. SASI enforcement of security policies: A retrospective. In *Proc. New Sec. Paradigms Workshop*, ACM (Caledon Hills, Ont., Sept. 1999), 87-94.

[21] Liu, P. and Jajodia, S. Multi-phase damage confinement in database systems for intrusion tolerance. In *Proc. Fourteenth Comput. Sec. Foundations Workshop*, IEEE (Cape Breton, N.S, June 2001), 191-205.

---

[a] Manuscript submitted December 28, 2001.

J. B. Michael, N. C. Rowe, and R. D. Riehle are with the Department of Computer Science, Naval Postgraduate School, 833 Dyer Rd., Monterey, Calif. 93943-5118 (e-mail: {bmichael, ncrowe, rdriehle}@nps.navy.mil).

M. Auguston is with the Department of Computer Science, New Mexico State University, Las Cruces, N.M. 88003-0001 (e-mail: mikau@cs.nmsu.edu).

[1] The registry is the repository of the local security policy used by the security reference manager and monitor to control access by threads to objects.