Mimicry Attacks on Host-Based Intrusion Detection Systems*

David Wagner
University of California, Berkeley
daw@cs.berkeley.edu

Paolo Soto
University of California, Berkeley
paolo@xcf.berkeley.edu

ABSTRACT

We examine several host-based anomaly detection systems and study their security against evasion attacks. First, we introduce the notion of a *mimicry attack*, which allows a sophisticated attacker to cloak their intrusion to avoid detection by the IDS. Then, we develop a theoretical framework for evaluating the security of an IDS against mimicry attacks. We show how to break the security of one published IDS with these methods, and we experimentally confirm the power of mimicry attacks by giving a worked example of an attack on a concrete IDS implementation. We conclude with a call for further research on intrusion detection from both attacker's and defender's viewpoints.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Host-based intrusion detection, anomaly detection, evasion attacks

General Terms

Security

1. INTRODUCTION

The goal of an intrusion detection system (IDS) is like that of a watchful burglar alarm: if an attacker manages to penetrate somehow our security perimeter, the IDS should set off alarms so that a system administrator may take appropriate action. Of course, attackers will not necessarily cooperate with us in this. Just as cat burglars use stealth to escape without being noticed, so too we can expect that computer hackers may take steps to hide their presence and try to evade detection. Hence if an IDS is to be useful, it would be a good idea to make it difficult for attackers to cause harm without being detected. In this paper, we study the ability of IDS's to reliably detect stealthy attackers who are trying to avoid notice.

The fundamental challenge is that attackers adapt in response to the defensive measures we deploy. It is not enough to design a system that can withstand those attacks that are common at the time the system is deployed. Rather, security is like a game of chess: one must anticipate all moves the attacker might make and ensure that the system will remain secure against all the attacker's possible responses. Consequently, an IDS that is susceptible to evasion attacks (where the attacker can cloak their attack to evade detection) is of uncertain utility over the long term: we can expect that if such an IDS sees widespread deployment, then attackers will change their behavior to routinely evade it. Since in practice many attacks arise from automated scripts, script writers may someday incorporate techniques designed to evade the popular IDS's in their scripts. In this sense, the very success of an approach for intrusion detection may lead to its own downfall, if the approach is not secure against evasion attacks.

Broadly speaking, there are two kinds of intrusion detection systems: network intrusion detection systems, and host-based intrusion detection systems. Several researchers have previously identified a number of evasion attacks on network intrusion detection systems [19, 18, 7, 1]. Motivated by those results, in this paper we turn our attention to host-based intrusion detection.

Though there has been a good deal of research on the security of network IDS's against evasion attacks, the security of host-based intrusion detection systems against evasion attacks seems not to have received much attention in the security literature. One can find many papers proposing new techniques for intrusion detection, and authors often try to measure their detection power by testing whether they can detect currently-popular attacks. However, the notion of security against adaptive adversarial attacks is much harder to measure, and apart from some recent work [23, 24], this subject does not seem to have received a great deal of coverage in the literature. To remedy this shortcoming, in this paper we undertake a systematic study of the issue.

Host-based intrusion detection systems can be further divided into two categories: signature-based schemes (i.e., misuse detection) and anomaly detection. Signature-based schemes are typically trivial to bypass simply by varying the attack slightly, much in the same way that polymorphic viruses evade virus checkers. We show in Section 4.2

*This research was supported in part by NSF CAREER CCR-0093337.

CCS'02, November 18–22, 2002, Washington, DC, USA.

how to automatically create many equivalent variants of a given attack, and this could be used by an attacker to avoid matching the IDS's signature of an attack. This is an unavoidable weakness of misuse detection. Evasion attacks on signature-based schemes are child's play, and so we do not consider them further in this paper.

Anomaly detection systems are more interesting from the point of view of evasion attacks, and in this paper we focus specifically on anomaly detection systems. We show in Section 3 several general evasion methods, including the notion of a mimicry attack and the idea of introducing "semantic no-ops" in the middle of the attack to throw the IDS off. Next, in Section 4, we introduce a principled framework for finding mimicry attacks, building on ideas from language and automata theory. We argue in Section 4.2 that nearly every system call can be used as a "no-op," giving the attacker great freedom in constructing an attack that will not trigger any intrusion alarms. Sections 5 and 6 describe our empirical experience in using mimicry attacks to escape detection: we convert an off-the-shelf exploit script into one that works without being detected by the pH IDS. Finally, in Sections 8 and 9 we conclude with a few parting thoughts on countermeasures and implications.

For expository purposes, this paper is written from the point of view of an attacker. Nonetheless, our goal is not to empower computer criminals, but rather to explore the limits of current intrusion detection technology and to enable development of more robust intrusion detection systems. The cryptographic community has benefitted tremendously from a combination of research on both attacks and defenses—for instance, it is now accepted wisdom that one must first become expert in codebreaking if one wants to be successful at codemaking, and many cryptosystems are validated according to their ability to stand up to concerted adversarial analysis—yet the intrusion detection community has not to date had the benefit of this style of adversarial scholarship. We hope that our work will help to jump-start such a dialogue in the intrusion detection research literature.

2. A TYPICAL HOST-BASED IDS

There have been many proposals for how to do host-based anomaly detection, but a paradigmatic (and seminal) example is the general approach of Forrest, et al. [3, 2, 8, 26, 21]. We will briefly review their scheme. They monitor the behavior of applications on the host by observing the interaction of those applications with the underlying operating system. In practice, security-relevant interactions typically take the form of system calls, and so their scheme works by examining the trace of system calls performed by each application.

Their scheme is motivated by using the human immune system as a biological analogy. If the system call traces of normal applications are self-similar, then we can attempt to build an IDS that learns the normal behavior of applications and recognizes possible attacks by looking for abnormalities. In the learning phase of this sort of scheme, the IDS gathers system call traces from times when the system is not under attack, extracts all subtraces containing six consecutive system calls, and creates a database of these observed subtraces[1]. A subtrace is deemed anomalous if it does not appear in this database. Then, in the monitoring phase, the abnormality of a new system call trace is measured by counting how many anomalous subtraces it contains.

The authors' experience is that attacks often appear as radically abnormal traces. For instance, imagine a mail client that is under attack by a script that exploits a buffer overrun, adds a backdoor to the password file, and spawns a new shell listening on port 80. In this case, the system call trace will probably contain a segment looking something like this:

> `open()`, `write()`, `close()`, `socket()`, `bind()`, `listen()`, `accept()`, `read()`, `fork()`.

Since it seems unlikely that the mail client would normally open a file, bind to a network socket, and fork a child in immediate succession, the above sequence would likely contain several anomalous subtraces, and thus this attack would be easily detected.

We selected Somayaji and Forrest's pH intrusion detection system [21] for detailed analysis, mainly because it was the only system where full source code could be obtained for analysis. Many other proposals for host-based anomaly detection may be found in the literature [3, 2, 8, 26, 21, 5, 14, 15, 4, 26, 12, 13, 17, 27]. However, pH is fairly typical, in the sense that many host-based IDS's rely on recognizing attacks based on the traces they produce, be it traces of system calls, BSM audit events, or Unix commands. We will use pH as a motivating example throughout the paper, but we expect that our techniques will apply more generally to host-based intrusion detection systems based on detecting anomalies in sequences of events. For instance, it should be possible to use our approach to analyze systems based on system call sequences [3, 2, 8, 26, 5, 27], data mining [14, 15], neural networks [4], finite automata [17], hidden Markov models [26], and pattern matching in behavioral sequences [12, 13].

3. BUILDING BLOCKS FOR EVASION

Background. First, let us start with a few assumptions to simplify the analysis to follow. It seems natural to assume that the attacker knows how the IDS works. This seems unavoidable: If the IDS becomes popular and is deployed at many sites, it will be extremely difficult to prevent the source code to the IDS from leaking. As usual, security through obscurity is rarely a very reliable defense, and it seems natural to assume that the IDS algorithm will be available for inspection and study by attackers.

Similarly, if the IDS relies on a database of normal behavior, typically it will be straightforward for the attacker to predict some approximation to this database. The behavior of most system software depends primarily on the operating system version and configuration details, and when these variables are held constant, the normal databases produced on different machines should be quite similar. Hence, an attacker could readily obtain a useful approximation to the database on the target host by examining the normal databases found on several other hosts of the same type, retaining only program behaviors common to all those other databases, and using the result as our prediction of the normal database on the target host. Since in our attacks the

[1]In practice, pH uses lookahead pairs to reduce the size of the database. This only increases the set of system call traces allowed by pH.

attacker needs only an under-approximation to the normal database in use, this should suffice. Hence, it seems reasonable to assume that the database of normal behaviors is mostly (or entirely) known.

Moreover, we also assume that the attacker can silently take control of the application without being detected. This assumption is not always satisfied, but for many common attack vectors, the actual penetration leaves no trace in the system call trace. For instance, exploiting a buffer overrun vulnerability involves only a change in the control flow of the program, but does not itself cause any system calls to be invoked, and thus no syscall-based IDS can detect the buffer overrun itself. In general, attacks can be divided into a penetration phase (when the attacker takes control of the application and injects remote code) and a exploitation phase (when the attacker exploits his control of the application to bring harm to the rest of the system by executing the recently-injected foreign code), and most anomaly detection systems are based on detecting the harmful effects of the exploitation, not on detecting the penetration itself. Consequently, it seems reasonable to believe that many applications may contain vulnerabilities that allow attackers to secretly gain control of the application.

With that background, the remainder of this section describes six simple ideas for avoiding detection, in order of increasing sophistication and power. We presume that the attacker has a malicious sequence of actions that will cause harm and that he wants to have executed; his goal is to execute this sequence without being detected.

Slip under the radar. Our first evasion technique is based on trying to avoid causing any change whatsoever in the observable behavior of the application. A simple observation is that system call-based IDS's can only detect attacks by their signature in the system call trace of the application. If it is possible to cause harm to the system without issuing any system calls, then the IDS has no hope of detecting such an attack. For instance, on some old versions of Solaris it was possible to become root simply by triggering the divide-by-zero trap handler, and this does not involve any system calls. However, such OS vulnerabilities appear to be exceptionally rare. As a more general instance of this attack class, an attacker can usually cause the application to compute incorrect results. For instance, a compromised web browser might invisibly elide all headlines mentioning the Democratic party whenever the user visits any news site, or a compromised mailer might silently change the word "is" to "isn't" in every third email from the company's CEO.

There seems to be little that an IDS can do about this class of attacks. Fortunately, the harm that an attacker can do to the rest of the system without executing any system calls appears to be limited.

Be patient. A second technique for evading detection is simply to be patient: wait passively for a time when the malicious sequence will be accepted by the IDS as normal behavior, and then pause the application and insert the malicious sequence. Of course, the attacker can readily recognize when the sequence will be allowed simply by simulating the behavior of the IDS. Simulating the IDS should be easy, since by our discussion above there are no secrets in the IDS algorithm.

Moreover, it is straightforward for the attacker to retain control while allowing the application to execute its usual sequence of system calls. For instance, the attacker who takes control of an application could embed a Trojan horse by replacing all the library functions in the application's address space by modified code. The replacement implementation might behave just like the pre-existing library code, except that before returning to its caller each function could check whether the time is right to begin executing the malicious sequence. After this modification is completed, the attacker could return the flow of program control to the application, confident in the knowledge that he will retain the power to regain control at any time. There are many ways to accomplish this sort of parasitic infection, and there seems to be no defense against such an invasion.

There is one substantial constraint on the attacker, though. This attack assumes that there will come a time when the malicious sequence will be accepted; if not, the attacker gains nothing. Thus, the power of this attack is limited by the precision of the database of normal behavior.

Another limitation on the attacker is that, after the malicious sequence has been executed, resuming execution of the application may well lead to an abnormal system call trace. In such a case, only two choices immediately present themselves: we could allow the application to continue executing (thereby allowing the IDS to detect the attack, albeit after the harm has already been done), or we could freeze the application permanently (which is likely to be very noticeable and thus might attract attention). A slightly better strategy may be to cause the application to crash in some way that makes the crash appear to have come from an innocuous program bug rather than from a security violation. Since in practice many programs are rather buggy, system administrators are used to seeing coredumps or the Blue Screen of Death from time to time, and they may well ignore the crash. However, this strategy is not without risk for the attacker.

In short, a patient attacker is probably somewhat more dangerous than a naive, impatient attacker, but the attacker still has to get lucky to cause any harm, so in some scenarios the risk might be acceptable to defenders.

Be patient, but make your own luck. One way the attacker can improve upon passive patience is by loading the dice. There are typically many possible paths of execution through an application, each of which may lead to a slightly different system call trace, and this suggests an attack strategy: the attacker can look for the most favorable path of execution and nudge the application into following that path.

As an optimization, rather than embedding a Trojan horse and then allowing the application to execute normally, the attacker can discard the application entirely and simulate its presence. For example, the attacker can identify the most favorable path of execution, then synthetically construct the sequence of system calls that would be executed by this path and issue them directly, inserting his malicious sequence at the appropriate point. The analysis effort can all be pre-computed, and thus a stealthy attack might simply contain a sequence of hard-coded system calls that simulate the presence of the application for a while and then eventually execute the malicious sequence.

In fact, we can see there is no reason for the attacker to restrict himself to the feasible execution paths of the application. The attacker can even consider system call traces that could not possibly be output by any execution of the application, so long as those traces will be accepted as "normal" by the IDS. In other words, the attacker can examine the set of system call traces that won't trigger any alarms and look for one such trace where the malicious sequence can be safely inserted. Then, once such a path is identified, the attacker can simulate its execution as above and proceed to evade the IDS.

In essence, we are mimicking the behavior of the application, but with a malicious twist. To continue the biological analogy, a successful mimic will be recognized as "self" by the immune system and will not cause any alarms. For this reason, we dub this the *mimicry attack* [25]. This style of attack is very powerful, but it requires a careful examination of the IDS, and the attacker also has to somehow identify favorable traces. We will study this topic in greater detail in Section 4.

Replace system call parameters. Another observation is that most schemes completely ignore the arguments to the system call. For instance, an innocuous system call

```
open("/lib/libc.so", O_RDONLY)
```

looks indistinguishable (to the IDS) from the malicious call

```
open("/etc/shadow", O_RDWR).
```

The evasion technique, then, is obvious. If we want to write to the shadow password file, there is no need to wait for the application to open the shadow password file during normal execution. Instead, we may simply wait for the application to open any file whatsoever and then substitute our parameters (`"/etc/shadow", O_RDWR`) for the application's. This is apparently another form of mimicry attack.

As far as we can tell, almost all host-based intrusion detection systems completely ignore system call parameters and return values. The only exception we are aware of is Wagner and Dean's static IDS [25], and they look only at a small class of system call parameters, so parameter-replacement attacks may be very problematic for their scheme as well.

Insert no-ops. Another observation is that if there is no convenient way to insert the given malicious sequence into the application's system call stream, we can often vary the malicious sequence slightly by inserting "no-ops" into it. In this context, the term "no-op" indicates a system call with no effect, or whose effect is irrelevant to the goals of the attacker. Opening a non-existent file, opening a file and then immediately closing it, reading 0 bytes from an open file descriptor, and calling `getpid()` and discarding the result are all examples of likely no-ops. Note that even if the original malicious sequence will never be accepted by the IDS, some modified sequence with appropriate no-ops embedded might well be accepted without triggering alarms.

We show later in the paper (see Section 4.2 and Table 1) that, with only one or two exceptions, nearly every system call can be used as a "no-op." This gives the attacker great power, since he can pad out his desired malicious sequence out with other system calls chosen freely to maximize the chances of avoiding detection. One might expect intuitively that every system call that can be found in the normal database may become reachable with a mimicry attack by inserting appropriate no-ops; we develop partial evidence to support this intuition in Section 6.

Generate equivalent attacks. More generally, any way of generating variations on the malicious sequence without changing its effect gives the attacker an extra degree of freedom in trying to evade detection. One can imagine many ways to systematically create equivalent variations on a given malicious sequence. For instance, any call to `read()` on an open file descriptor can typically be replaced by a call to `mmap()` followed by a memory access. As another example, in many cases the system calls in the malicious sequence can be re-ordered. An attacker can try many such possibilities to see if any of them can be inserted into a compromised application without detection, and this entire computation can be done offline in a single precomputation.

Also, a few system calls give the attacker special power, if they can be executed without detection as part of the exploit sequence. For instance, most IDS's handle `fork()` by cloning the IDS and monitoring both the child and parent application process independently. Hence, if an attacker can reach the `fork()` system call and can split the exploit sequence into two concurrent chunks (e.g., overwriting the password file and placing a backdoor in the `ls` program), then the attacker can call `fork()` and then execute the first chunk in the parent and the second chunk in the child. As another example, the ability to execute the `execve()` system call gives the attacker the power to run any program whatsoever on the system.

Of course, the above ideas for evasion can be combined freely. This makes the situation appear rather grim for the defenders: The attacker has many options, and though checking all these options may require a lot of effort on the attacker's part, it also seems unclear whether the defenders can evaluate in advance whether any of these might work against a given IDS. We shall address this issue next.

4. A THEORETICAL FRAMEWORK

In this section, we develop a systematic framework for methodically identifying potential mimicry attacks. We start with a given malicious sequence of system calls, and a model of the intrusion detection system. The goal is to identify whether there is any trace of system calls that is accepted by the IDS (without triggering any alarms) and yet contains the malicious sequence, or some equivalent variant on it.

This can be formalized as follows. Let Σ denote the set of system calls, and Σ^* the set of sequences over the alphabet Σ. We say that a system call trace $T \in \Sigma^*$ is *accepted* (or *allowed*) by the IDS if executing the sequence $T = \langle T_1, T_2, \ldots \rangle$ does not trigger any alarms. Let $\mathcal{A} \subseteq \Sigma^*$ denote the set of system call traces allowed by the IDS, i.e.,

$$\mathcal{A} \stackrel{\text{def}}{=} \{T \in \Sigma^* : T \text{ is accepted by the IDS}\}.$$

Also, let $\mathcal{M} \subseteq \Sigma^*$ denote the set of traces that achieve the attacker's goals, e.g.,

$$\mathcal{M} \stackrel{\text{def}}{=} \{T \in \Sigma^* \;:\; T \text{ is an equivalent variant on the given malicious sequence}\}.$$

Now we can succinctly state the condition for the existence of mimicry attacks. The set $\mathcal{A} \cap \mathcal{M}$ is exactly the set of traces that permit the attacker to achieve his goals without detection, and thus mimicry attacks are possible if and only if $\mathcal{A} \cap \mathcal{M} \neq \emptyset$. If the intersection is non-empty, then any of its elements gives a stealthy exploit sequence that can be used to achieve the intruder's goals while reliably evading detection.

The main idea of the proposed analytic method is to frame this problem in terms of formal language theory. In this paper, $\mathcal{A}$ is a regular language. This is fairly natural [20], as finite-state IDS's can always be described as finite-state automata and thus accept a regular language of syscall traces. Moreover, we insist that $\mathcal{M}$ also be a regular language. This requires a bit more justification (see Section 4.2 below), but hopefully it does not sound too unreasonable at this point. It is easy to generalize this framework still further[2], but this formulation has been more than adequate for all the host-based IDS's considered in our experiments.

With this formulation, testing for mimicry attacks can be done automatically and in polynomial time. It is a standard theorem of language theory that if L, L' are two regular languages, then so is $L \cap L'$, and $L \cap L'$ can be computed effectively [11, §3.2]. Also, given a regular language L'', we can efficiently test whether $L'' \stackrel{?}{=} \emptyset$, and if L'' is non-empty, we can quickly find a member of L'' [11, §3.3]. From this, it follows that if we can compute descriptions of $\mathcal{A}$ and $\mathcal{M}$, we can efficiently test for the existence of mimicry attacks. In the remainder of this section, we will describe first how to compute $\mathcal{A}$ and then how to compute $\mathcal{M}$.

4.1 Modelling the IDS

In Forrest's IDS, to predict whether the next system call will be allowed, we only need to know the previous five system calls. This is a consequence of the fact that Forrest's IDS works by looking at all subtraces of six consecutive system calls, checking that each observed subtrace is in the database of allowable subtraces.

Consequently, in this case we can model the IDS as a finite-state automaton with statespace given by five-tuples of system calls and with a transition for each allowable system call action. More formally, the statespace is $Q = \Sigma^5$ (recall that Σ denotes the set of system calls), and we have a transition

$$(s_0, s_1, s_2, s_3, s_4) \xrightarrow{s_5} (s_1, s_2, s_3, s_4, s_5)$$

for each subtrace $(s_0, \ldots, s_5)$ found in the IDS's database of allowable subtraces. The automaton can be represented efficiently in the same way that the normal database is represented.

Next we need a initial state and a set of final (accepting) states, and this will require patching things up a bit. We introduce a new absorbing state Alarm with a self-transition $\mathsf{Alarm} \xrightarrow{s} \mathsf{Alarm}$ on each system call $s \in \Sigma$, and we ensure that every trace that sets off an intrusion alarm ends up in the state Alarm by adding a transition $(s_0, s_1, s_2, s_3, s_4) \xrightarrow{s_5} \mathsf{Alarm}$ for each subtrace $(s_0, \ldots, s_5)$ that is *not* found in the IDS's database of allowable subtraces. Then the final (accepting) states are all the non-alarm states, excluding only the special state Alarm.

The initial state of the automaton represents the state the application is in when the application is first penetrated. This is heavily dependent on the application and the attack vector used, and presumably each different vulnerability will lead to a different initial state. For instance, if there is a buffer overrun that allows the attacker to gain control just after the application has executed five consecutive `read()` system calls, then the initial state of the automaton should be $(\mathtt{read}, \mathtt{read}, \mathtt{read}, \mathtt{read}, \mathtt{read})$.

Extensions. In practice, one may want to refine the model further to account for additional features of the IDS. For instance, the locality frame count, which is slightly more forgiving of occasional mismatched subtraces and only triggers alarms if sufficiently many mismatches are seen, can be handled within a finite-state model. For details, see Appendix A.

4.2 Modelling the malicious sequence

Next, we consider how to express the desired malicious sequence within our framework, and in particular, how to generate many equivalent variations on it. The ability to generate equivalent variations is critical to the success of our attack, and rests on knowledge of equivalences induced by the operating system semantics. In the following, let $M = \langle M_1, \ldots, M_n \rangle \in \Sigma^*$ denote a malicious sequence we want to sneak by the IDS.

Adding no-ops. We noted before that one simple way to generate equivalent variants is by freely inserting "no-ops" into the malicious sequence M. A "no-op" is a system call that has no effect, or more generally, one that has no effect on the success of the malicious sequence M. For instance, we can call `getpid()` and ignore the return value, or call `brk()` and ignore the newly allocated returned memory, and so on.

A useful trick for finding no-ops is that we can invoke a system call with an invalid argument. When the system call fails, no action will have been taken, yet to the IDS it will appear that this system call was executed. To give a few examples, we can `open()` a non-existent pathname, or we can call `mkdir()` with an invalid pointer (say, a NULL pointer, or one that will cause an access violation), or we can call `dup()` with an invalid file descriptor. Every IDS known to the authors ignores the return value from system calls, and this allows the intruder to nullify the effect of a system call while fooling the IDS into thinking that the system call succeeded.

The conclusion from our analysis is that almost every system call can be nullified in this way. Any side-effect-free system call is already a no-op. Any system call that takes a pointer, memory address, file descriptor, signal number, pid, uid, or gid can be nullified by passing invalid arguments. One notable exception is `exit()`, which kills the process no matter what its argument is. See Table 1 for a list of all system calls we have found that might cause difficulties for the attacker; all the rest may be freely used to generate equivalent variants on the malicious sequence[3]. The surprise is not how hard it is to find nullifiable system calls, but rather how easy it is to find them—with only a few exceptions, nearly every system call is readily nullifiable. This gives the attacker extraordinary freedom to vary the malicious exploit sequence.

We can characterize the equivalent sequences obtained this way with a simple regular expression. Let $\mathcal{N} \subseteq \Sigma$ denote the set of nullifiable system calls. Consider the regular

[2] For instance, we could allow $\mathcal{A}$ or $\mathcal{M}$ (but not both) to be context-free languages without doing any violence to the polynomial-time nature of our analysis.

[3] It is certainly possible that we might have overlooked some other problematic system calls, particularly on systems other than Linux. However, we have not yet encountered any problematic system call not found in Table 1.

System call	Nullifiable?	Useful to an attacker?	Comments
`exit()`	No	No	Kills the process, which will cause problems for the intruder.
`pause()`	No	Unlikely	Puts the process to sleep, which would cause a problem for the intruder. Attacker might be able to cause process to receive signal and wake up again (e.g., by sending SIGURG with TCP out of band data), but this is application-dependent.
`vhangup()`	No	Usually	Linux-specific. Hangs up the current terminal, which might be problematic for the intruder. But it is very rarely used in applications, hence shouldn't cause a problem.
`fork()`	No	Usually	Creates a new copy of the process. Since the IDS will probably clone itself to monitor each separately, this is unlikely to cause any problems for the attacker. (Similar comments apply to `vfork()` and to `clone()` on Linux.)
`alarm()`	No	Usually	Calling `alarm(0)` sets no new alarms, and will likely be safe. It does have the side-effect of cancelling any previous alarm, which might occasionally interfere with normal application operation, but this should be rare.
`setsid()`	No	Usually	Creates a new session for this process, if it is not already a session leader. Seems unlikely to interfere with typical attack goals in practice.
`socket()`	Yes	Yes	Nullify by passing socket type parameter.
`pipe()`	Yes	Yes	Nullify by passing NULL pointer parameter.
`open()`	Yes	Yes	Nullify by passing NULL filename parameter.
...	Yes	Yes	...

Table 1: A few system calls and whether they can be used to build equivalent variants of a given malicious sequence. The second column indicates whether the system call can be reliably turned into a "no-op" (i.e., nullified), and the third column indicates whether an attacker can intersperse this system call freely in a given malicious sequence to obtain equivalent variants. For instance, `exit()` is not nullifiable and kills the process, hence it is not usable for generating equivalent variants of a malicious sequence. This table shows all the system calls we know of that an attacker might not be able to nullify; the remaining system calls not shown here are easily nullified.

expression defined by

$$\mathcal{M} \stackrel{\text{def}}{=} \mathcal{N}^* M_1 \mathcal{N}^* M_2 \mathcal{N}^* \cdots \mathcal{N}^* M_n \mathcal{N}^*.$$

This matches the set of sequences obtained from M by inserting no-ops, and any sequence matching this regular expression will have the same effect as M and hence will be interchangeable with M. Moreover, this regular expression may be expressed as a finite-state automaton by standard methods [11, §2.8], and in this way we obtain a representation of the set $\mathcal{M}$ defined earlier, as desired.

Extensions. If necessary, we could introduce further variability into the set of variants considered by considering equivalent system calls. For instance, if a `read()` system call appears in the malicious sequence M, we could also easily replace the `read()` with a `mmap()` system call if this helps avoid detection. As another example, we can often collapse multiple consecutive `read()` calls into a single `read()` call, or multiple `chdir()` system calls into a single `chdir()`, and so on.

All of these equivalences can also be modelled within our finite-state framework. Assume we have a relation R on $\Sigma^* \times \Sigma^*$ obeying the following condition: if $X, X' \in \Sigma^*$ satisfy $R(X, X')$, then we may assume that the sequence X can be equivalently replaced by X' without altering the resulting effect on the system. Suppose moreover that this relation can be expressed by a finite-state transducer, e.g., a Mealy or Moore machine; equivalently, assume that R forms a rational transduction. Define

$$\mathcal{M} \stackrel{\text{def}}{=} \{X' \in \Sigma^* : R(M, X') \text{ holds}\}.$$

By a standard result in language theory [11, §11.2], we find that $\mathcal{M}$ is a regular language, and moreover we can easily compute a representation of $\mathcal{M}$ as a finite-state automaton given a finite-state representation of R.

Note also that this generalizes the strategy of inserting no-ops. We can define a relation $R_{\mathcal{N}}$ by $R_{\mathcal{N}}(X, X')$ if X' is obtained from X by inserting no-ops from the set $\mathcal{N}$, and it is not hard to see that the relation $R_{\mathcal{N}}$ can be given by a finite-state transduction. Hence the idea of introducing no-ops can be seen as a special case of the general theory based on rational transductions.

In summary, we see that the framework is fairly general, and we can expect to model both the IDS and the set of malicious sequences as finite-state automata.

5. IMPLEMENTATION

We implemented these ideas as follows. First, we trained the IDS and programmatically built the automaton $\mathcal{A}$ from the resulting database of normal sequences of system calls. The automaton $\mathcal{M}$ is formed as described above.

The next step is to form the composition of $\mathcal{A}$ and $\mathcal{M}$ by taking the usual product construction. Our implementation tests for a non-empty intersection by constructing the product automaton $\mathcal{A} \times \mathcal{M}$ explicitly in memory [11, §3.2] and performing a depth-first search from the initial state to see if any accepting state is reachable [11, §3.3]; if yes, then we've found a stealthy malicious sequence, and if not, the mimicry attack failed. In essence, this is a simple way of model-checking the system $\mathcal{A}$ against the property $\mathcal{M}$.

We note that there are many ways to optimize this computation by using ideas from the model-checking literature. For instance, rather than explicitly computing the entire product automaton in advance and storing it in memory, to reduce space we could perform the depth-first search generating states lazily on the fly. Also, we could use hashing to keep a bit-vector of previously visited states to further reduce memory consumption [9, 10]. If this is not enough, we could even use techniques from symbolic model-checking to represent the automata $\mathcal{A}$ and $\mathcal{M}$ using BDD's and then compute their product symbolically with standard algorithms [16].

However, we have found that these fancy optimizations seem unnecessary in practice. The simple approach seems adequate for the cases we've looked at: in our experiments, our algorithm runs in less than a second. This is not surprising when one considers that, in our usage scenarios, the automaton $\mathcal{A}$ typically has a few thousand states and $\mathcal{M}$ contains a half dozen or so states, hence their composition contains only a few tens of thousands of states and is easy to compute with.

6. EMPIRICAL EXPERIENCE

In this section, we report on experimental evidence for the power of mimicry attacks. We investigated a number of host-based anomaly detection systems. Although many papers have been written proposing various techniques, we found only one working implementation with source code that we could download and use in our tests: the pH (for process homeostasis) system [21]. pH is a derivative of Forrest, et al.'s early system, with the twist that pH responds to attacks by slowing down the application in addition to raising alarms for the system administrator. For each system call, pH delays the response by 2^m time units, where m counts the number of mismatched length-6 subtraces in the last 128 system calls. We used pH version 0.17 running on a fresh Linux Redhat 5.0 installation with a version 2.2.19 kernel[4]. Our test host was disconnected from the network for the duration of our experiments to avoid the possibility of attacks from external sources corrupting the experiment.

We also selected an off-the-shelf exploit to see whether it could be made stealthy using our techniques. We chose one more or less at random, selecting an attack script called `autowux.c` that exploits the "site exec" vulnerability in the `wuftpd` FTP server. The `autowux` attack script exploits a format string vulnerability, and it then calls `setreuid(0,0)`, escapes from any chroot protection, and execs `/bin/sh` using the `execve()` system call. It turns out that this is a fairly typical payload: the same shellcode can be found in many other attack scripts that exploit other, unrelated vulnerabilities[5]. We conjecture that the authors of the `autowux` script just copied this shellcode from some previous source, rather than developing new shellcode. Our version of Linux Redhat 5.0 runs `wuftpd` version `wu-2.4.2-academ[BETA-15](1)`, and we trained pH by running `wuftpd` on hundreds of large file downloads over a period of two days. We verified that pH detects the unmodified exploit[6].

Next, we attempted to modify the exploit to evade detection. We parsed pH's database of learned length-6 subtraces and built an automaton $\mathcal{A}$ recognizing exactly those system call traces that never cause any mismatches. We did not bother to refine this representation to model the fact that intruder can safely cause a few occasional mismatches without causing problems (see Appendix A), as such a refinement turned out to be unnecessary. Also, we examined the point in time where `autowux` mounts its buffer overflow attack against the `wuftpd` server. We found that the window of the last five system calls executed by `wuftpd` is

(`fstat()`, `mmap()`, `lseek()`, `close()`, `write()`)

when the exploit first gains control. This determines the initial state of $\mathcal{A}$.

In addition, we reverse engineered the exploit script and learned that it performs the following sequence of 15 system calls:

`setreuid(0,0)`, `dup2(1,2)`, `mkdir("sh")`, `chroot("sh")`, $9 \times$ `chdir("..")`, `chroot("/")`, `execve("/bin/sh")`.

We noticed that the nine consecutive `chdir("..")` calls can, in this case, be collapsed into a single

`chdir("../../../../../../../../../")`.

As always, one can also freely introduce no-ops. With these two simple observations, we built an automaton $\mathcal{M}$ recognizing the regular expression

$\mathcal{N}^*$ `setreuid()` $\mathcal{N}^*$ `dup2()` $\mathcal{N}^*$ `mkdir()` $\mathcal{N}^*$ `chroot()` $\mathcal{N}^*$ `chdir()` $\mathcal{N}^*$ `chroot()` $\mathcal{N}^*$ `execve()` $\mathcal{N}^*$.

Our program performs a depth-first search in the product automaton $\mathcal{A} \times \mathcal{M}$ and informs us that $\mathcal{A} \cap \mathcal{M} = \emptyset$, hence there is no stealthy trace matching the above regular expression.

Next, we modified the attack sequence slightly by hand to repair this deficiency. After interactively invoking our tool a few times, we discovered the reason why the original pattern was infeasible: there is no path through the normal database reaching `dup2()`, `mkdir()`, or `execve()`, hence no attack that uses any of these system calls can completely avoid mismatches. However, we note that these three system calls can be readily dispensed with. There is no need to create a new directory; an existing directory will do just as well in escaping from the chroot jail, and as a side benefit will leave fewer traces. Also, the `dup2()` and `execve()` are needed only to spawn an interactive shell, yet an attacker can still cause harm by simply hard-coding in the exploit shellcode the actions he wants to take without ever spawning a shell. We hypothesized that a typical harmful action an attacker might want to perform is to add a backdoor root account into the password file, hence we proposed that an attacker might be just as happy to perform the following

[4]Since this work was done, version 0.18 of pH has been released. The new version uses a longer window of length 9, which might improve security. We did not test whether this change improves the resistance of pH to mimicry attacks.

[5]It is interesting and instructive to notice that such a widespread attack payload includes provisions by default to always attempt escaping from a chroot jail. The lesson is that, if a weak protection measure becomes widespread enough, eventually attackers will routinely incorporate countermeasures into all their attacks. The implications for intrusion detection systems that are susceptible to mimicry attacks are troubling.

[6]We took care to ensure that the IDS did not learn the exploit code as "normal" in the process. All of our subsequent experiments were on a virgin database, trained from scratch using the same procedure and completely untouched by any attack.

```
read() write() close() munmap() sigprocmask() wait4()
sigprocmask() sigaction() alarm() time() stat() read()
alarm() sigprocmask() setreuid() fstat() getpid()
time() write() time() getpid() sigaction() socketcall()
sigaction() close() flock() getpid() lseek() read()
kill() lseek() flock() sigaction() alarm() time()
stat() write() open() fstat() mmap() read() open()
fstat() mmap() read() close() munmap() brk() fcntl()
setregid() open() fcntl() chroot() chdir() setreuid()
lstat() lstat() lstat() lstat() open() fcntl() fstat()
lseek() getdents() fcntl() fstat() lseek() getdents()
close() write() time() open() fstat() mmap() read()
close() munmap() brk() fcntl() setregid() open() fcntl()
chroot() chdir() setreuid() lstat() lstat() lstat()
lstat() open() fcntl() brk() fstat() lseek() getdents()
lseek() getdents() time() stat() write() time() open()
getpid() sigaction() socketcall() sigaction() umask()
sigaction() alarm() time() stat() read() alarm()
getrlimit() pipe() fork() fcntl() fstat() mmap() lseek()
close() brk() time() getpid() sigaction() socketcall()
sigaction() chdir() sigaction() sigaction() write()
munmap() munmap() munmap() exit()
```

Figure 1: A stealthy attack sequence found by our tool. This exploit sequence, intended to be executed after taking control of `wuftpd` through the "site exec" format string vulnerability, is a modification of a pre-existing sequence found in the `autowux` exploit. We have underlined the system calls from the original attack sequence. Our tool takes the underlined system calls as input, and outputs the entire sequence. The non-underlined system calls are intended to be nullified: they play the role of "semantic no-ops," and are present only to ensure that the pH IDS does not detect our attack. The effect of the resulting stealthy exploit is to escape from a chroot jail and add a backdoor root account to the system password file.

variant on the original exploit sequence:

```
setreuid(0,0), chroot("pub"),
 chdir("../../../../../../../../../"), chroot("/"),
 open("/etc/passwd", O_APPEND|O_WRONLY),
 write(fd, "toor:AAaaaaaaaaaaa:0:0::/:/bin/sh", 33),
 close(fd), exit(0)
```

where `fd` represents the file descriptor returned by the `open()` call (this value can be readily predicted). The modified attack sequence becomes root, escapes from the chroot jail, and appends a backdoor root account to the password file. To check whether this modified attack sequence could be executed stealthily, we built an automaton $\mathcal{M}$ recognizing the regular expression

$$\mathcal{N}^* \texttt{ setreuid() } \mathcal{N}^* \texttt{ chroot() } \mathcal{N}^* \texttt{ chdir() } \mathcal{N}^* \texttt{ chroot() } \mathcal{N}^* \texttt{ open() } \mathcal{N}^* \texttt{ write() } \mathcal{N}^* \texttt{ close() } \mathcal{N}^* \texttt{ exit() } \mathcal{N}^*.$$

We found a sequence that raises no alarms and matches this pattern. See Fig. 1 for the stealthy sequence. Finding this stealthy sequence took us only a few hours of interactive exploration with our search program, once the software was implemented.

We did not build a modified exploit script to implement this attack. Instead, to independently verify the correctness of the stealthy sequence, we separately ran this sequence through `stide`[7] and confirmed that it would be accepted with zero mismatches by the database generated earlier. Note that we were able to transform the original attack sequence into a modified variant that would not trigger even a single mismatch but that would have a similarly harmful effect. In other words, there was no need to take advantage of the fact that pH allows a few occasional mismatches without setting off alarms: our attack would be successful no matter what setting is chosen for the pH locality frame count threshold. This makes our successful results all the more meaningful.

In summary, our experiments indicate that sophisticated attackers can evade the pH IDS. We were fairly surprised at the success of the mimicry attack at converting the `autowux` script into one that would avoid detection. On first glance, we were worried that we would not be able to do much with this attack script, as its payload contains a fairly unusual-looking system call sequence. Nonetheless, it seems that the database of normal system call sequences is rich enough to allow the attacker considerable power.

Shortcomings. We are aware of several significant limitations in our experimental methodology. We have not compiled the stealthy sequence in Fig. 1 into a modified exploit script or tried running such a modified script against a machine protected by pH. Moreover, we assumed that we could modify the `autowux` exploit sequence so long as this does not affect the effect of a successful attack; however, our example would have been more convincing if the attack did not require modifications to the original exploit sequence.

Also, we tested only a single exploit script (`autowux`), a single vulnerable application (`wuftpd`), a single operating system (Redhat Linux), a single system configuration (the default Redhat 5.0 installation), and a single intrusion detection system (pH). This is enough to establish the presence of a risk, but it does not provide enough data to assess the magnitude of the risk or to evaluate how differences in operating systems or configurations might affect the risk.

We have not tried to assess how practical the attack might be. We did not study how much effort or knowledge is required from an attacker to mount this sort of attack. We did not empirically test how effectively one can predict the configuration and IDS normal database found on the target host, and we did not measure whether database diversity is a significant barrier to attack. We did not estimate what percentage of vulnerabilities would both give the attacker sufficient control over the application to mount a mimicry attack and permit injection of enough foreign code to execute the entire stealthy sequence. Also, attacks often get better over time, and so it may be too soon to draw any definite conclusions. Because of all these unknown factors, more thorough study will be needed before we can confidently evaluate the level of risk associated with mimicry attacks in practice.

[7] Because pH uses lookahead pairs, `stide` is more restrictive than pH. However, the results of the test are still valid: since our modified sequence is accepted by `stide`, we can expect that it will be accepted by pH, too. If anything, using `stide` makes our experiment all the more meaningful, as it indicates that `stide`-based IDS's will also be vulnerable to mimicry attacks.

7. RELATED WORK

There has been some other recent research into the security of host-based anomaly detection systems against sophisticated, adaptive adversaries.

Wagner and Dean briefly sketched the idea of mimicry attacks in earlier work [25, §6]. Giffin, Jha, and Miller elaborated on this by outlining a metric for susceptibility to evasion attacks based on attack automata [6, §4.5]. Somayaji suggested that it may be possible in principle, but difficult in practice, to evade the pH IDS, giving a brief example to justify this claim [22, §7.5]. None of these papers developed these ideas in depth or examined the implications for the field, but they set the stage for future research.

More recently, and independently, Tan, Killourhy, and Maxion provided a much more thorough treatment of the issue [23]. Their research shows how attackers can render host-based IDS's blind to the presence of their attacks, and they presented compelling experimental results to illustrate the risk. In follow-up work, Tan, McHugh, and Killourhy refined the technique and gave further experimental confirmation of the risk from such attacks [24]. Their methods are different from those given in this paper, but their results are in agreement with ours.

8. DISCUSSION

Several lessons suggest themselves after these experiments. First and foremost, where possible, intrusion detection systems should be designed to resist mimicry attacks and other stealthy behavior from sophisticated attackers. Our attacks also give some specific guidance to IDS designers. It might help for IDS's to observe not only what system calls are attempted but also which ones fail and what error codes are returned. It might be a good idea to monitor and predict not only which systems calls are executed but also what arguments are passed; otherwise, the attacker might have too much leeway. Moreover, the database of normal behavior should be as minimal and precise as possible, to reduce the degree of freedom afforded to an attacker.

Second, we recommend that all future published work proposing new IDS designs include a detailed analysis of the proposal's security against evasion attacks. Even if this type of vulnerability cannot be completely countered through clever design, it seems worthwhile to evaluate carefully the risks.

Finally, we encourage IDS designers to publicly release a full implementation of their designs, to enable independent security analysis. There were several proposed intrusion detection techniques we would have liked to examine in detail for this work, but we were unable to do so because we did not have access to a reference implementation.

9. CONCLUSIONS

We have shown how attackers may be able to evade detection in host-based anomaly intrusion detection systems, and we have presented initial evidence that some IDS's may be vulnerable. It is not clear how serious a threat mimicry attacks will be in practice. Nonetheless, the lesson is that it is not enough to merely protect against today's attacks: one must also defend against tomorrow's attacks, keeping in mind that tomorrow's attackers might adapt in response to the protection measures we deploy today. We suggest that more attention could be paid in the intrusion detection community to security against adaptive attackers, and we hope that this will stimulate further research in this area.

10. ACKNOWLEDGEMENTS

We thank Umesh Shankar, Anil Somayaji, and the anonymous reviewers for many insightful comments on an earlier draft of this paper. Also, we are indebted to Somayaji for making the pH source code publicly available, without which this research would not have been possible.

11. REFERENCES

[1] M. Chung, N. Puketza, R.A. Olsson, B. Mukherjee, "Simulating Concurrent Intrusions for Testing Intrusion Detection Systems: Parallelizing Intrusions," *National Information Systems Security Conference*, pp.173–183, 1995.

[2] S. Forrest, S.A. Hofmeyr, A. Somayaji, T. A. Longstaff, "A Sense of Self for Unix Processes," *1996 IEEE Symposium on Security & Privacy.*

[3] S. Forrest, A.S. Perelson, L. Allen, R. Cherukuri, "Self-Nonself Discrimination in a Computer," *1994 IEEE Symposium on Security & Privacy.*

[4] A.K. Ghosh, A. Schwartzbard, M. Schatz, "Learning Program Behavior Profiles for Intrusion Detection," *1st USENIX Workshop on Intrusion Detection & Networking Monitoring*, 1999.

[5] A.K. Ghosh, A. Schwartzbard, M. Schatz, "Using Program Behavior Profiles for Intrusion Detection," *3rd SANS Workshop on Intrusion Detection & Response*, 1999.

[6] J.T. Giffin, S. Jha, B.P. Miller, "Detecting Manipulated Remote Call Streams," *11th USENIX Security Symposium*, 2002.

[7] M. Handley, C. Kreibich, V. Paxson, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," *10th USENIX Security Symposium*, 2001.

[8] S. Hofmeyr, S. Forrest, A. Somayaji, "Intrusion Detection Using Sequences of System Calls," *Journal of Computer Security*, vol. 6, pp. 151-180, 1998.

[9] G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, 1990.

[10] G.J. Holzmann, "The Model Checker Spin," *IEEE Trans. on Software Engineering*, Special issue on Formal Methods in Software Practice, May 1997.

[11] J.E. Hopcroft, J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[12] T. Lane, C.E. Brodley, "Sequence Matching and Learning in Anomaly Detection for Computer Security," *AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, pp.49–49, 1997.

[13] T. Lane, C.E. Brodley, "Temporal Sequence Learning and Data Reduction for Anomaly Detection," *ACM Trans. Information & System Security*, vol. 2, no. 3, pp.295–331, 1999.

[14] W. Lee, S.J. Stolfo, "Data Mining Approaches for Intrusion Detection," *7th USENIX Security Symposium*, 1998.

[15] W. Lee, S.J. Stolfo, K. Mok, "A Data Mining Framework for Building Intrusion Detection Models,"

IEEE Symposium on Security & Privacy, 1999.

[16] K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[17] C. Michael, A. Ghosh, "Using Finite Automata to Mine Execution Data for Intrusion Detection: A Preliminary Report," *RAID 2000*, LNCS 1907, pp.66–79, 2000.

[18] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks*, 31(23–24), pp.2435–2463, 14 Dec. 1999.

[19] T.H. Ptacek, T.N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Jan. 1998.

[20] F. Schneider, "Enforceable security policies," *ACM Transactions on Information & System Security*, vol. 3, no. 1, pp.30–50, Feb. 2000.

[21] A. Somayaji, S. Forrest, "Automated Response Using System-Call Delays," *9th Usenix Security Symposium*, pp.185–197, 2000.

[22] A.B. Somayaji, "Operating System Stability and Security through Process Homeostasis," Ph.D. dissertation, Univ. New Mexico, Jul. 2002.

[23] K.M.C. Tan, K.S. Killourhy, R.A. Maxion, "Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits," to appear at *RAID 2002*, 16–18 Oct. 2002.

[24] K. Tan, J. McHugh, K. Killourhy, "Hiding Intrusions: From the abnormal to the normal and beyond," to appear at *5th Information Hiding Workshop*, 7–9 Oct. 2002.

[25] D. Wagner, D. Dean, "Intrusion Detection via Static Analysis," *IEEE Symposium on Security & Privacy*, 2001.

[26] C. Warrender, S. Forrest, B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," *1999 IEEE Symposium on Security & Privacy*.

[27] A. Wespi, M. Dacier, H. Debar, "Intrusion Detection Using Variable-Length Audit Trail Patterns," *RAID 2000*, LNCS 1907, pp.110–129, 2000.

APPENDIX

A. EXTENSIONS: HOW TO MODEL MORE SOPHISTICATED IDS'S

Forrest, et al., have proposed [2] counting the total number of mismatched length-6 subtraces and only triggering an alarm if the total mismatch count exceeds some threshold, say 7 mismatches. This gives the intruder an extra degree of freedom, because now the exploit code is free to cause a few mismatches, as long as they are not too numerous.

We can easily extend our finite-state model above to account for this degree of freedom, as follows. We add an extra dimension to the statespace, counting the number of mismatches so far. Thus, the statespace becomes $Q' \stackrel{\text{def}}{=} (Q \times \{0, 1, \ldots, 6\}) \cup \{\mathsf{Alarm}\}$, and each non-alarm state is a pair (q, m) of a state $q \in Q = \Sigma^5$ from the old model and a count $m \in \{0, \ldots, 6\}$ of the number of mismatches seen so far. For each non-alarm transition $q \xrightarrow{s} q'$ of the old model, we introduce transitions $(q, m) \xrightarrow{s} (q', m)$ for each $m = 0, 1, \ldots, 6$. Also, for each alarm transition $(s_0, \ldots, s_4) \xrightarrow{s} \mathsf{Alarm}$ of the old model, we introduce transitions $((s_0, \ldots, s_4), m) \xrightarrow{s} ((s_1, \ldots, s_4, s), m + 1)$ for each $m = 0, 1, \ldots, 6$, where we view the notation $(q, 7)$ as short-hand for the Alarm state. As usual, we introduce self-loops from Alarm to itself on each system call, and the accepting states are exactly the non-alarm states. Note that the size of the automaton has increased by only a small constant factor, hence this transformation should be practical.

As another example, Warrender, et al., propose a slightly different extension, which is more forgiving of occasional mismatches [26]. They suggest that the IDS should trigger an alarm only if at least 6 of the last 20 length-6 subtraces are mismatches. We can model this extension within our finite-state framework by adding an extra dimension to the statespace to account for the history of the last 20 subtraces. Let $W \stackrel{\text{def}}{=} \{S \subseteq \{1, 2, \ldots, 20\} : |S| \leq 6\}$ denote the set of all subsets of $\{1, 2, \ldots, 20\}$ of cardinality at most 6. Then the statespace of our refined finite-state automaton becomes $Q' \stackrel{\text{def}}{=} (Q \times W) \cup \{\mathsf{Alarm}\}$, and each non-alarm state is a pair $(q, \{i_1, \ldots, i_k\})$ of a state $q \in Q = \Sigma^5$ from the unadorned model and a list $i_1, \ldots, i_k$ of times in the recent past where a mismatch was found. If S is a set of integers, let $\text{delay}(S)$ denote the set $\text{delay}(S) \stackrel{\text{def}}{=} \{s + 1 : s \in S\} \cap \{1, \ldots, 20\}$. For each non-alarm transition $q \xrightarrow{s} q'$ of the old model, we introduce transitions $(q, S) \xrightarrow{s} (q', \text{delay}(S))$ for each $S \in W$. Also, for each alarm transition $(s_0, \ldots, s_4) \xrightarrow{s} \mathsf{Alarm}$ of the old model, we introduce transitions $((s_0, \ldots, s_4), m) \xrightarrow{s} ((s_1, \ldots, s_4, s), \text{delay}(S) \cup \{1\})$ for each $S \in W$. Here we view the notation (q, S) as short-hand for the Alarm state when $S \notin W$, i.e., when $|S| > 6$. This transformation does increase the size of the automaton by a noticeable amount: namely, by a factor of $\binom{20}{6} + \cdots + \binom{20}{0} = 60460$.

In short, we can see that many natural extensions and variations on the basic IDS scheme can be incorporated within our framework. We conclude that the idea of modelling an IDS as a finite-state automaton seems broadly applicable.