What Happened to Anomaly Detection?*

Hajime Inoue Anil Somayaji
School of Computer Science
Carleton University
{hinoue, soma}@ccsl.carleton.ca

December 7, 2006

Abstract

Mimicry attacks have motivated much of the recent work in program-level intrusion detection. Whether based on runtime monitoring or static analysis, work by Kruegel (2003), Giffin (2005) and others have focused on the challenge posed by attacks that are designed to emulate normal program behavior. Although mimicry attacks can be a significant threat in certain circumstances, we argue that they are not more significant than issues such as false positives and overall attack coverage. Furthermore, given the inevitable compromises involved in creating intrusion detection systems, a focus on mimicry attacks almost necessitates a compromise on these other qualities. In this paper we present a simple model of program-level intrusion detection systems that captures these trade-offs, and we apply this model to the work of Forrest (1996), Giffin, Kruegel, and Sekar (2001). This analysis shows that advances in mimicry resistance have been accompanied by an overall reduction in performance. We then propose an alternative interpretation of mimicry attacks that suggests improved designs for next generation systems.

1 Introduction

The continuing susceptibility of modern computer platforms to security threats has motivated work on "post-hoc" security mechanisms. One strategy has been to detect attacks as they occur. Although systems such as virus scanners and firewalls can be said to serve such a role, the standard design paradigm for such approaches is knows as intrusion detection systems (IDSs) [7]. IDSs can be classified by how they detect attacks: either using specifiers of known security threats (signature-based detection), specifiers of legal activity (specification or model-based detection), or through deviations from past observed behavior (anomaly detection).

Anomaly detection holds great promise as an approach to intrusion detection. Unlike most signature-based systems, anomaly detection is capable of detecting zero-day attacks; unlike specification-based systems, anomaly detection requires minimal manual work for creating profiles. To date, however, anomaly detection systems suffer from three basic limitations:

1. They require training, ideally within the deployed environment.
2. Legitimate changes in system usage can produce false alarms.

*Carleton University School of Computer Science TR-07-09

3. They fail to detect some security violations.

All approaches to intrusion detection are fallible, making (3) a general concern; anomaly detection systems, however, have been specifically criticized for being susceptible to “mimicry attacks,” or attacks in which malicious behavior masquerades as normal behavior [30].

Many researchers have studied mimicry attacks, particularly within the domain of program-level intrusion detection, both on the side of developing better ways to mount mimicry attacks [31, 27, 29, 18, 12], and in terms of developing detection methods that are resistant (or immune) to mimicry attacks [9, 11, 19, 20] . Such arms races of better attacks and defenses are established strategies for improving security mechanisms, particularly in the realm of cryptography. Unfortunately, in the field of program-level intrusion detection, this arms race has, in our view, led to the development of problematic intrusion detection strategies.

Developers of new intrusion detection methods have created anomaly detection methods that require larger amounts of training and are worse at accommodating legitimate changes in normal behavior, or they have proposed alternatives to program-level anomaly detection that cannot detect classes of attacks, such as embedded Trojan code and flawed program logic, that can be detected through anomaly detection. Resistance to mimicry attacks is a worthy goal; however, given the limited scope of mimicry attacks, particularly in the program monitoring domain, it is not worth compromising overall detection precision or attack coverage.

The rest of this paper proceeds as follows. We first review the recent history of detection proposals, attacks, and counter-proposals in the program-level IDS literature. Next, we present a simple set-theoretic framework for understanding program-level intrusion detection. We then use this framework to specifically analyze the proposals of Forrest [10, 25], Sekar [24], Kruegel [19], and Giffin [11], in order to better understand their overall operational characteristics and their tolerance for mimicry attacks. We conclude with suggestions for future research.

2 Mimicry Attacks and Responses

Program-level intrusion detection systems monitor program behavior at runtime with the goal of detecting security violations as they occur. In principle any aspect of program behavior could be modelled; most work, however, has focused on monitoring the system call behavior of running processes on UNIX-like systems, in accordance with the original framing of the problem by Forrest [10].

Mimicry attacks against program-level IDSs were first suggested by Wagner and Dean in 2001 [30].[1] Their attack strategy hinges on the fact that all intrusion detection systems make generalizations about the behavior of the systems they are monitoring. In a mimicry attack against a program-level IDS, an attacker crafts their exploit such that the targeted program's behavior will appear to be legitimate with respect to an IDS's profile of program behavior.

This formulation of mimicry attacks has gained wide attention and many groups have focused on mimicry attacks as a major flaw in anomaly intrusion detection systems; it has also inspired others to create program-level IDSs that are resistant to mimicry attacks. In this section we first describe Forrest's **stide** [16] and the mimicry attacks that target it. We then discuss more recent work on mimicry resistant systems.

[1]Ptacek and Newsham earlier discussed mimicry attacks in networks in 1998 [23].

2.1 System Call Sequence Mimicry Attacks

The primary target for research on mimicry attacks has been the system call profiling technique proposed by Forrest [10]. In their approach to anomaly detection, a profile of normal program behavior is produced by first monitoring the system calls executed by a process. These kernel invocations are then used to construct a linear trace of system calls, where each call is represented by a small integer. A short window, typically between 6 and 10 system calls in length, is then slid across the trace. The resulting profile is either just the record of the observed short sequences [15], or it stored as a simple abstraction of the observed sequences that they refer to as "lookahead pairs" [10]. In either case, after a profile is constructed, anomalous program behavior is signalled when the monitored program executes a short sequence of system calls that is not present in the training profile. A tree-based implementation of the pure sequence-based profiling method was released in the **stide** software package [16], along with the system call traces from their original 1996 paper [10].

Wagner and Dean were the first to point out the potential problems of using system call sequences to detect security violations:

> Note that imprecise models contain impossible paths, which introduces a vulnerability to mimicry attacks if any of those paths can reach an insecure state. Consequently, the primary defense against mimicry attacks lies in high-precision models [30].

In this context, "impossible paths" are program paths which their static analysis method deems legal but are impossible in the program text. More generally, mimicry attacks are based upon an attacker creating code paths that are not present in the original binary but are permitted by a given profile of program behavior. Thus, we can paraphrase Wagner and Dean as follows: Because generalizations allow the possibility of mimicry attacks, decreasing generality reduces those possibilities.[2]

Wagner and Dean's paper was more concerned with the false positive problem of anomaly IDS's than with mimicry attacks; they simply pointed out mimicry attacks as a potential problem. Since then, however, there have been several papers on the feasibility of mimicry attacks on the **stide** anomaly detector. Tan and Maxion in 2002 [28] studied the sensitivity of **stide** to the choice of window size using the datasets from Forrest's original paper [10]. This analysis formed the basis of the first implementations of mimicry attacks by Tan and others [27, 29]. Shortly thereafter, Wagner and Soto [31] demonstrated an exploit against a more mature system using the same type of detector (pH) [26, 25]. Recently, Kruegel [18] and Giffin [12] have independently developed automated approaches to mimicry attacks on **stide**.

Although there are differences, the basic procedure used to develop mimicry attacks is similar in each of these attack methods. A mimicry attack on **stide** requires two things, a working code injection attack (such as a buffer overflow attack) and the targeted application's normal profile. The normal profile is easily obtained dynamically (one can use a number of system call loggers), or it can be obtained through static analysis by extracting execution paths that must be followed by the targeted application in normal operation. Then, one tailors the attack by inserting extraneous system calls, which are usually crafted to do nothing, into the attack payload until all of the attack system call sequences are present in the normal profile. When the suitably modified code injection attack is then mounted on a targeted application, **stide** will signal no anomalies because every

[2]Recently the term model-based systems has been used to describe IDSs that use static-analysis generated profiles of normal behavior. We adopt that terminology in this paper for static approaches but use "normal profile" for anomaly-based systems, which learn profiles using observed behavior.

observed system call will appear to be normal—even though the actual code being executed is not present in the original program text.

2.2 Mimicry-Resistant IDSs

The limitations of **stide**, both in terms of observed false positives and susceptibility to mimicry attacks, have motivated work on alternative approaches to program-level intrusion detection. The main response has been to add execution context to the normal profile. Others have added different features to improve precision.

Much of this work on mimicry resistance has built upon the work of Sekar [24]. Sekar suggested using a finite automaton that linked system calls to the program counter value of the initiating instruction. Although their motivation was to improve the performance in comparison to system-call sequence anomaly detection, this technique is the basis of other approaches designed to prevent mimicry attacks. Linking the system call to its address forces a mimic to construct more complicated return-into-application attacks, which are variants of return-into-others attacks [21]. Such attacks, while difficult, are possible, as demonstrated by Kruegel [18].

Feng and his colleagues added call-stack information to system-call sequences in their 2003 paper [9]. Essentially, this was increasing the level of inspection of the stack from the top frame, as in Sekar's case, to a set of enclosing contexts. By increasing the granularity of the execution state space, they hoped to eliminate mimicry attacks and "impossible path" exploits. Using this extra information they were able to detect both the attacks described by Wagner as well as a more sophisticated one that tries to fool naive call-stack systems by manipulating return information.

This technique is further developed by Giffin in a paper that combined multiple stack contexts with static analysis [11]. They eliminate most "impossible path" attack possibilities, including those available in their earlier paper [9], by making even intra-function control jumps monitorable. Because their static analysis technique is able to capture all possible paths of execution (and not just the ones observed during a set of program executions), their technique also eliminates false positives, an advantage they cite for their method relative to dynamic monitoring techniques.

Others have attempted to prevent mimicry attacks by having the kernel protect key information. Linn [20] use statically generated model of program behavior, like Giffin [11]; however, they move the model into the kernel, so that the model cannot be manipulated. They also obfuscate the instructions that trap into the kernel. They then monitor many kinds of interrupts, including errors, as an alternative to the standard system call trap instructions. These changes make it difficult for exploits to scan the address space to find an appropriate return-into-others attack.

Xu's "waypoint" idea is also similar to Giffin's in that they add call-stack information to the anomaly feature set [32]. The biggest difference is that addresses are not associated with system calls. Instead, waypoints express a statically determined system call permission set. By allowing the kernel to keep an audited copy of the call stack, exploits must use system calls sequences that are in the normal profile of the current execution context (so-called local mimicry exploits). This change prevents the possibility that exploits can falsify their execution contexts, albeit at the cost of much more frequent traps into the kernel. "Program Shepherding," introduced by Kiriansky and his colleagues [17], is somewhat similar to the waypoint idea; however, the implementation is very different. Shepherding hosts applications in a virtual machine that makes certain that every branch is implied by the program text at load time. Thus, system calls need not be guarded. Barrantes's system, which also uses a VM, has similar properties [4].

Kruegel, instead of combining additional features to system calls, concentrated on the arguments to system calls to defeat mimicry attacks [19]. Mimicry attacks are constructed by inserting system calls with no or irrelevant side effects. This often produces system calls with very odd arguments. Using this observation, Kruegel's system is able to detect the mimicry attack demonstrated by Wagner and Soto [31]. They do not make the claim they are able to detect all mimicry attacks because they are only looking at arguments. They suggest combining a sequence-call detector to their system would eliminate much of the mimicry attack space.

Giffin and colleagues presented an alternative way of restricting the arguments to system calls through static analysis. Specifically, they can restrict execution by fixing some system call arguments if that information can be determined using information available at compile and at program load time [13].

3 Analysis

From the discussion of the last section, two trends in the responses to mimicry attacks should be apparent: the use of additional features of program execution and the move from models derived from observed dynamic behavior to models inferred through static analysis. Features were added to monitoring systems to decrease the possibility of mimicry attacks. Model inference was added to mitigate the increased probability of false positives. To better understand the impact of these shifts, we now present a formal model describing the properties of the profile spaces of anomaly and model-based IDSs in the program behavior domain.

3.1 Profile Spaces

Let Σ^* be the set of all possible execution states. Each element in this set consists of the state of the registers, the devices, and memory. A perfect IDS would be able to observe all features of execution space and divide Σ^* into **safe** and $\overline{\textbf{safe}}$ sets. No mimicry attacks would be possible with such an IDS. In practice we do not know how to precisely distinguish between **safe** and $\overline{\textbf{safe}}$; the errors that arise when practical IDSs attempt to distinguish between these sets result in missed attacks (false negatives) and false alarms (false positives).

Because it is not computationally feasible to observe all aspects of program behavior, practical IDSs instead observe at most a few features of program execution. Thus, their distinguishable execution spaces are much smaller than Σ^*. Each element of this space is a set itself, whose elements are subsets of Σ^*. Each profile, whether determined through model inference or training, is a subset of this new space (call it $\textbf{safe}'$). The $\textbf{safe}'$ set is analogous to **safe**. Although the union of all elements in $\textbf{safe}'$ are in Σ^*, $\textbf{safe}' \neq \textbf{safe}$. That is why all IDSs are vulnerable to mimicry and other attacks. In this formalization, mimicry attacks exist because:

$$|\bigcup \textbf{safe}' - \textbf{safe}| > 0$$

That is, there are execution states labelled as safe in $\textbf{safe}'$ that are not in **safe**.

Let us further investigate mimicry attacks within this formalization. Consider two hypothetical systems using the same feature set, one anomaly-based, one model-based. These systems are depicted in a Venn diagram in in Figure 1. Mimicry attacks are the intersection of attacks with the imperfect safe set of each system. Let us call the model-based system's safe set $\textbf{safe}_m$ and the anomaly-based system's safe set $\textbf{safe}_a$. The model-based system is always susceptible to at least as many mimicry attacks as the anomaly-based system because $\textbf{safe}_m$ generalizes over all

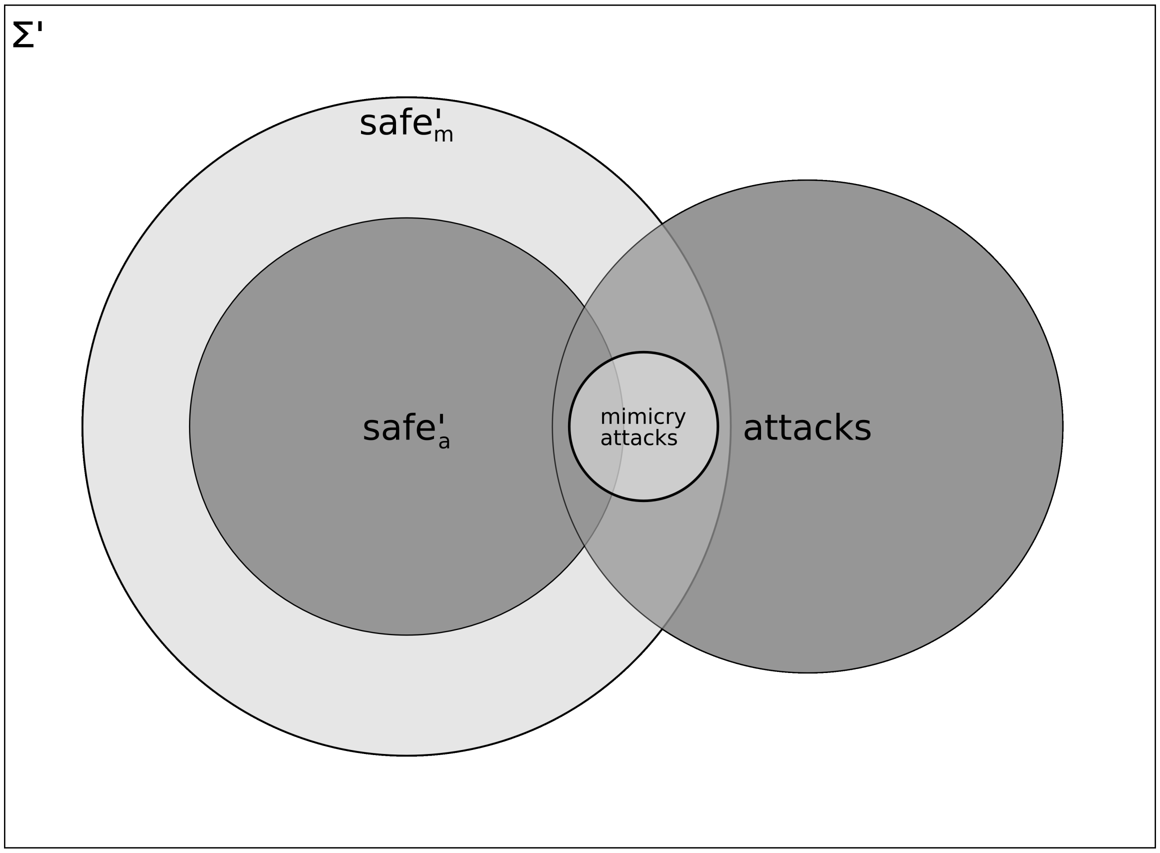

Figure 1: A Venn diagram comparing the **safe** sets of a model and anomaly-based system observing the same feature sets. The execution space shown is the generalized execution space Σ', not Σ^*. Using the same feature set, $\mathbf{safe}'_a \subset \mathbf{safe}'_m$. Attacks are shown as a set intersecting both **safe** sets. Mimicry attacks are the class of attacks shown by a subset of the intersection of a system's **safe** set with **attacks**. Under these conditions, a model-based system is always susceptible to at least as many mimicry attacks as an anomaly-based system.

inputs.[3]. This result contradicts experience: the model-based system described in Section 2 are less vulnerable to mimicry attacks. The difference is the result of the model-based systems being more precise—they use larger numbers of features (see Section 3.3).

Systems that use different numbers of features are best compared in Σ^* space. Figure 2 shows hypothetical model and anomaly-based system with the model IDS using a larger set of features. Note that here there is potentially a significant difference between a given model and anomaly-based system. Choosing a method that is best able to approximate **safe** is the key challenge for designers of program-level intrusion detection systems.

[3]The model-based systems described above generalize over more than just inputs to make the static analysis feasible

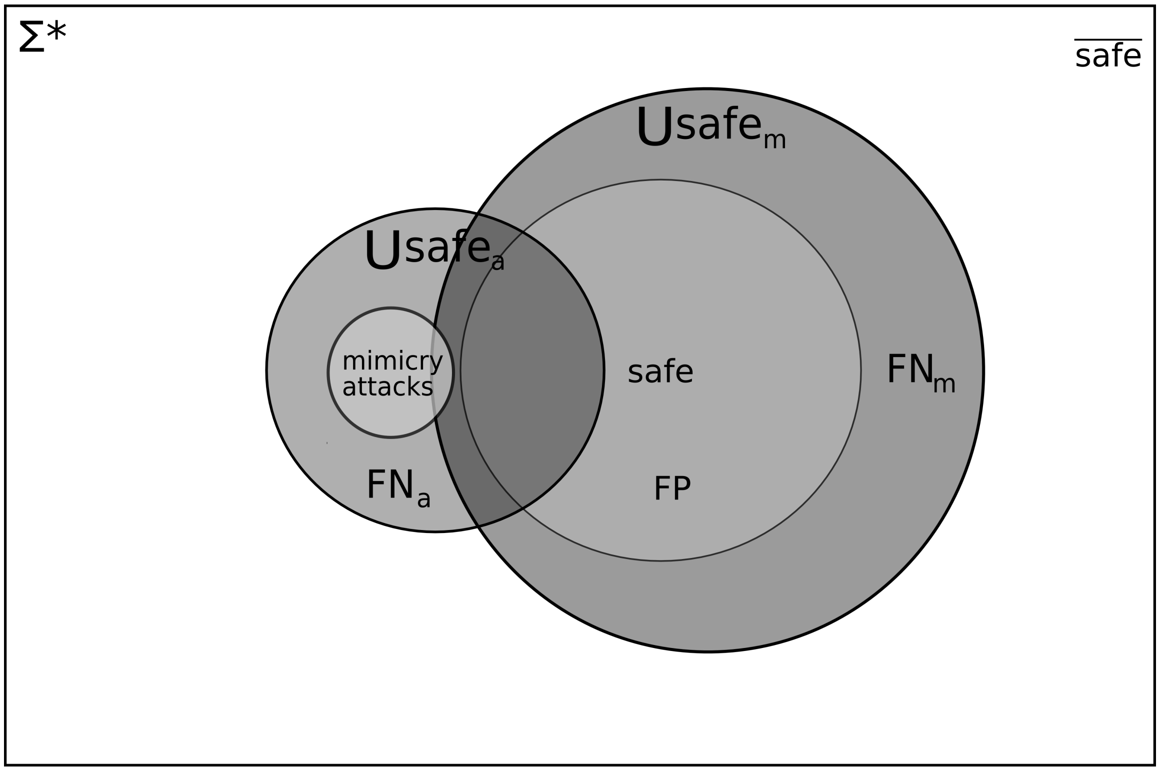

Figure 2: Another Venn diagram comparing **safe** with normal profiles of a model-based system ($\bigcup \textbf{safe}_m$) and an anomaly-based system ($\bigcup \textbf{safe}_a$). These systems do not use the same feature set, thus $\textbf{safe}_a$ and $\textbf{safe}_m$ are not in the same space and $\textbf{safe}_a \not\subset \textbf{safe}_m$. Mimicry attacks are depicted here as a subset within $\bigcup \textbf{safe}_a$. False positives (FP), are execution states that are within **safe** but are not in the $\textbf{safe}_a$ or $\textbf{safe}_m$. False negatives (FN), or execution states reflecting attacks that are not detected by the IDSs, are labelled FN_m and FN_a for false negative by the model-based system and the anomaly-based system, respectively.

3.2 False Positives

One key distinction that is apparent from Figure 2 is that model-based systems do not have false positives while anomaly systems do. Model-based systems of program behavior generalize over legal inputs and permit all program paths that are legitimately encoded in the program binary; thus, it is virtually impossible for a program to deviate from its model in a non-attack scenario. Anomaly-based systems, however, only generalize over the inputs seen during training, capturing just the code paths induced by those inputs. False positives are produced, then, when new inputs trigger previously unobserved, but legitimate behavior.

$$|\textbf{safe} - \bigcup \textbf{safe}_m| = 0$$
$$|\textbf{safe} - \bigcup \textbf{safe}_a| > 0$$

The lack of false positives, of course, is the fundamental advantage of model-based systems. The more execution features an IDS employs, the more a model-based approach is preferred, because a larger Σ requires more training for an anomaly-based system to eliminate false positives.

Consider the systems discussed in Section 2 in light of this formalization. Good exemplars of systems are Forrest's lookahead pairs [10], Sekar's automata IDS [24], Kruegel's system call argument IDS [19], and Giffin's model-based IDS [11].

We can explicitly state these sizes:

$$\begin{aligned} |\Sigma^s| &= ac \\ |\Sigma^f| &= wc^2 \\ |\Sigma^g| &= cn^h \end{aligned}$$

Here s stands for Sekar, f for Forrest, and g for Giffin, while a is the number of addresses within a program that invoke a system call, c is the number of system calls, h is the maximum stack depth, n is the number of sites that invoke a system-call invoking function, and w is the window size. Depending on the window size and the complexity of the application, the sizes of Σ^s and Σ^f can be similar. It is clear, however, that Σ^g is much larger. Based upon this analysis, we can order the sizes of their execution spaces:

$$|\Sigma^s| < |\Sigma^f| < |\Sigma^g| < |\Sigma^*|$$

Kruegel's system uses a variety of different spaces, so we cannot compare it directly. However, because it uses values of arguments to system calls as well as the calls themselves, several of the Kruegel's execution spaces are much larger than those of Forrest's and Sekar's systems.

Although it can be difficult to determine from published data, the false positive rates for the anomaly detection systems appear to roughly correspond to the relative sizes of their execution spaces (especially when the data from Somayaji [25] is taken into account). Giffin's system, although it has a larger space than Forrest's or Sekar's, does not have false positives due to its generalization over program inputs.

3.3 False Negatives

Let us now consider the systems susceptibility to successful attacks, or false negatives. It is clear that many types of mimicry attacks are more difficult in Giffin's system than in Sekar's or Forrest's. Kruegel's system, because it generalizes less, also shows resistance to these attacks. The ability to defeat mimicry attacks, however, does not imply an overall better ability to detect security violations. In our formalization, false negatives are characterized by the elements that are not safe that are classified as safe through generalization in each of the systems:

$$\begin{aligned} \text{attacks}_m &= \bigcup \mathbf{safe}_m - \overline{\mathbf{safe}} \\ \text{attacks}_a &= \bigcup \mathbf{safe}_a - \overline{\mathbf{safe}} \end{aligned}$$

For any fixed feature set, $|\text{attacks}_m| \geq |\text{attacks}_a|$, because $\mathbf{safe}_a \subset \mathbf{safe}_m$.

Model-based IDSs eliminate many mimicry attacks, but they cannot detect any attack that does not change the control-flow graph of the application. There are several types of attacks that do this, however. Among them are attacks based upon vulnerabilities, such as authentication flaws or embedded Trojan code, that can be exploited using "legitimate" code paths. Further, attacks based upon flaws in security policy, race conditions, and interacting features in general cannot be detected by low-level, statically-derived models of program behavior.

To see examples of such attacks, we do not have to look beyond Forrest's original paper [10]. In this paper most of the attacks were not based upon code injection vulnerabilities; instead, they relied upon coding errors and unneeded, dangerous features. For example, the sunsendmailcp exploit [3, 14] relied on the ability of an attacker to pass a command-line option to sendmail that would confuse it to the point that it didn't know where to deliver a mail message. As a last resort, it would attempt to deliver the message to `/usr/tmp/dead.letter`. An attacker, however, would have already created a symbolic link in this location. The compromised sendmail would thus follow this symbolic link and overwrite any file on the system with the attacker's email message. Note that this attack relied upon pre-existing code in the sendmail application; thus, model-based IDSs would not be able to detect this attack.

Most of the other attacks follow a similar pattern. The lprcp attack [2] relied upon a feature interaction problem between the use of only 1000 unique temporary spool files and the seldom-used features of queuing but not printing a file referred to by a symbolic link. The sendmail decode exploit made use of a rarely used mail alias that processed incoming email messages through the uudecode command. In fact, of the tested exploits, only one could have been reasonably subject to a mimicry attack: the buffer overflow exploit of the syslog code in sendmail. For the same reason, this is the one successful attack from Forrest's paper that could have been reasonably detected by a model-based system.

Recently, it has been suggested by Chen [5] that system-call based IDSs cannot detect attacks that do not directly modify the control flow graph of a targeted program. Chen is correct about model-based systems but mischaracterizes anomaly-based systems. Anomaly-based systems are generally not able to detect the modification of memory locations directly; rather, they detect the changes in program behavior caused by those modifications. Even non-statistical systems such as **stide** implicitly keep track of what code paths have been executed; thus, a data modification attack can be detected so long as at some point it causes a program to execute an unusual code path.

Note that this quality also allows program-level anomaly detection systems to detect embedded Trojan code. For example, Somayaji and Forrest showed that system call monitoring methods could detect a backdoor in ssh [1], even if the code for that backdoor was present when the anomaly detection system was trained [26]. In contrast, when model-based systems can detect Trojan code, they require that the model have been generated using a pristine copy of the program.

The differences in ability of anomaly and model-based systems to detect intrusions come from the inclusion of extra elements in model-based systems **safe** set. Alternatively, one can view these differences as one-bit of frequency information per element in that set. The key insight, though, is that by abandoning learning and instead inferring the notion of normal from the program text, model-based systems, while effective at suppressing mimicry attacks, are unable to detect many other important attack classes.

4 Discussion

In the world of practical security, circumventable protection mechanisms are the rule, not the exception. Virus scanners and network signature-based IDSs can only reliably detect attacks for which they have specific rules; minor modifications of an attack can bypass these systems. Firewalls are regularly traversed by malicious code. Buffer overflow protections such as non-executable stacks [8], stack canaries [6], and address space randomization [22] can all be bypassed by a determined attacker. All of these defenses, however, are now widely deployed on production systems because they provide reliable protection against significant threats without placing too much of a burden on users or system administrators.

Anomaly detection systems have not achieved anywhere near the level of acceptance of these other post-hoc security mechanisms. This lack of adoption, however, is arguably not due to their potential susceptibility to mimicry attacks, but rather to their requirements for training and the expectation of high false positive rates. The potential advantages of anomaly detection, however, remain compelling: the ability to detect zero-day attacks, easy adaptability to new programs and environments, and the freedom from signature updates are all features that would be useful to many individuals and organizations.

Proposed anomaly detection methods such as those by Forrest [10] and Sekar [24] appear to have the potential to fulfill many of the requirements of a deployable anomaly detection system. Despite some follow-up work [25], though, we still know relatively little about how such systems perform in production environments. This lack of testing is critical because we have no firm theoretical basis for comparing anomaly detection systems; feature space sizes are not enough to properly compare observed features or learning methods.

Although mimicry attacks may have seemed to be a good criteria for comparing intrusion detection systems, the limitations of model-based IDSs illustrate the problems with such a focus: systems that have the lowest false positive rates and the highest resistance to mimicry attacks no longer have the qualities that make anomaly detection a worthwhile strategy in the first place. Note that we are not arguing that model-based program IDSs will not find their place in deployed systems; rather, we see that model-based systems would complement, not replace, anomaly detection-based IDSs.

For future work, then, we see the problem of anomaly-based IDSs in simple terms: we need to better understand $\mathbf{safe}_a$ relative to **safe**, so that we can maximize the overlap between these sets. Rather than develop a single system that is based on many complex features, we believe that **safe** can be better approximated through multiple simple dynamic profiles based upon different observables. Each profile can be individually optimized to reduce false positives, potentially at the cost of increasing false negatives including mimicry attacks; maximum attack coverage would then come through integrating their outputs.

Theoretical and empirical work on such profile learning methods will potentially lead to program-level anomaly detection methods that produce fewer false alarms and detect a wider range of attacks. We hope this paper provides a firm basis and motivation for such future advances.

5 Conclusion

Since the release of Forrest's IDS using system call sequences [10], efforts have been made to attack and improve upon it. Mimicry attacks have motivated researchers to model other features such as call stack information and system call arguments, and the higher rates of observed false positives of such systems led to the development of model-based IDSs.

We have shown, using a formalization of execution spaces, why anomaly detection systems using more complex features are more susceptible to false positives. We have also shown how model-based systems are vulnerable to a larger class of attacks that anomaly-based systems can detect.

Anomaly detection is often criticized for its false positives and training requirements. We believe that instead of looking for alternative methods, research should concentrate directly on mitigating these disadvantages through the development and combination of simple models based upon multiple observed features of program behavior.

6 Acknowledgements

This work was supported by the Discovery grant program of Canada's National Sciences and Engineering Research Council (NSERC) and MITACS.

References

[1] Root kit ssh 5.0. Website, January 2000. `http:www.ne.jp/asahi/linux/timecop/`.

[2] [8LGM]. [8lgm]-advisory-3.unix.lpr.19-aug-1991. Website, 1991. `http://www.alw.nih.gov/Security/8lgm/8lgm-Advisory-03.html`.

[3] [8LGM]. [8lgm]-advisory-16.unix.sendmail-6-dec-1994. Website, 1994. `http://www.8lgm.org/advisories/\%5B8lgm\%5D-Advisory-16.UNIX.sendmail-6-Dec-1994.html`.

[4] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 281–289, New York, NY, USA, 2003. ACM Press.

[5] Shuo Chen, Jun Xu, and Emre C. Sezer. Non-control-data attacks are realistic threats. In *14th Annual Usenix Security Symposium*, Aug 2005.

[6] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

[7] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.

[8] Solar Designer. Linux kernel patch from the openwall project. http://www.openwall.com/linux/, 2001.

[9] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[10] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 120, Washington, DC, USA, 1996. IEEE Computer Society.

[11] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th Annual Network and Distributed Systems Security Symposium (NDSS)*, Feb 2004.

[12] J. T. Giffin, S. Jha, and B. Miller. Automated discovery of mimicry attacks. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID '06)*, Sep 2006.

[13] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller. Environment-sensitive intrusion detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID '05)*. Springer, Sep 2005.

[14] Markus Hűbner and Zhart. The ultimate sendmail hole list. Website, 1996. `http://bau2.uibk.ac.at/matic/buglist.htm`.

[15] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3), 1998.

[16] Steven A. Hofmeyr and Julie Rehmeyr. Stide: Sequence time-delay embedding, 1998.

[17] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium (Security '02)*, Aug 2002.

[18] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *14th Annual Usenix Security Symposium*, Aug 2006.

[19] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Gionvanni Vigna. On the detection of anomalous system call arguments. In *Proceeding of ESORICS 2003*, pages 326–343. Springer-Verlag Berlin Heidelberg, oct 2003.

[20] C.M. Linn, M. Rajagolpalan, S. Baker, C. Collberg, S.K. Debray, and J.H. Hartman. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Security Symposium*, pages 239–254, 2005.

[21] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 11(58), Dec 2001.

[22] PaX Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[23] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Jan 1998.

[24] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[25] Anil Somayaji. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, University of New Mexico, 2002.

[26] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 14–17, 2000.

[27] K. M. C. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID '02)*, 2002.

[28] Kymie M. C. Tan and Roy A. Maxion. "why 6?" defining the operational limits of stide, an anomaly-based intrusion detector. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 188, Washington, DC, USA, 2002. IEEE Computer Society.

[29] Kymie M. C. Tan, John McHugh, and Kevin S. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *IH '02: Revised Papers from the 5th International Workshop on Information Hiding*, pages 1–17, London, UK, 2003. Springer-Verlag.

[30] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–169, 2001.

[31] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM Press.

[32] Haizhi Xu, Wenliang Du, and Steve J. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection (RAID '04)*, 2004.