Security and Science of Agility

Patrick McDaniel†, Trent Jaeger†, Thomas F. La Porta†, Nicolas Papernot†
Robert J. Walls†, Alexander Kott‡, Lisa Marvel‡, Ananthram Swami‡
Prasant Mohapatra±, Srikanth V. Krishnamurthy∓, Iulian Neamtiu∓
†Department of Computer Science and Engineering, Penn State University
‡United States Army Research Laboratory, Adelphi, Maryland
±Department of Computer Science, University of California Davis
∓Department of Computer Science and Engineering, University of California Riverside

Moving target defenses alter the environment in response to adversarial action and perceived threats. Such defenses are a specific example of a broader class of system management techniques called *system agility*. In its fullest generality, agility is any reasoned modification to a system or environment in response to a functional, performance, or security need. This paper details a recently launched 10-year Cyber-Security Collaborative Research Alliance effort focused in-part on the development of a new science of system agility, of which moving target defenses are a central theme. In this context, the consortium seeks to address the questions of when, what, and how to employ changes to improve the security of an environment, as well as consider how to measure and weigh the effectiveness of different approaches to agility. We discuss several fundamental challenges in developing and using MTD maneuvers, and outline several broad classes of mechanisms that can be used to implement them. We conclude by detailing specific MTD mechanisms used to adaptively quarantine vulnerable code in Android applications, and consider ways of comparing cost and payout of its use.

Category and Subject Descriptors
D.4.6 [**OPERATING SYSTEMS**]: Security and Protection

General Terms Agility, Moving Target Defenses

1. INTRODUCTION

Moving target defenses (MTDs) are a specific example of a larger security and management technique called *agility*. Agility is a reasoned modification to a system or environment in response to a functional, performance, or security need. Agility can be as simple as changing a small configuration parameter or as complex as reconfiguring the topology of a network. Indeed, agility is common in many contexts, e.g., cloud systems often allocate resources and services on demand as needs dictate. Here, agility services seek to realize new computational structures in which environments alter their structure and function in response to users and system needs.

Illustrating one important application of system agility, moving target defenses change the environment to ensure survivability, mitigate or deceive adversaries, or increase detection capabilities. More concretely, MTDs are *maneuvers* executed by a human or autonomous defender in an attempt to increase the security of an environment. This paper introduces a new effort aimed at developing a science for securing environments via agility maneuvers. This effort seeks to develop a science that addresses the questions of when, what, and how to employ changes to improve the security of an environment while maintaining the availability, functionality, and performance of its elements.

The remainder of this paper considers a science of system agility using MTD as the central motivating example. We begin in the next section by considering the challenges to be considered when developing a generalized framework for system agility. In particular, we (*a*) explore the need and practice of hiding MTD behavior while sustaining system function, (*b*) consider hidden dependencies and impacts of MTD maneuvers on other systems, and (*c*) highlight the tradeoffs of MTD in terms of performance, function, and security. We also discuss how to develop models used to characterize MTD strategies based on game theory.

Next, we introduce generalized classes of mechanisms for agility that can be applied to a broad range of environments. We introduce three agility abstractions that can be applied to a great many domains. Wrappers are classes of agility mechanisms that intercept and rewrite inputs to security-sensitive interfaces, e.g., system calls, network traffic, and application-specific APIs. System capabilities are generalized system features/interfaces that can be enabled, altered or disabled at run-time, e.g., enabling scripting within a browser, or permitting remote procedures calls. Distributors are services that allocate, place, re-locate, and disable services in response to evolving needs and other environmental factors. The broad behavior of these facilities is presented and challenges considered.

We conclude by illustrating many of the challenges and mechanisms highlighted throughout in a single example. This example–a mechanism for detecting and altering Android application behavior–presents one point in the spectrum of moving target defenses. Here, the application environment changes its interface by sealing off code with security vulnerabilities. This discussion explores the structure of such a service, as well as considers the ways in which the cost and payouts can be measured and compared.

1.1 CSec Collaborative Research Alliance

The agility work described throughout is being pursued within the Cyber Security Research Alliance (CSec CRA). The CSec CRA is a consortium of academic, industry, and government research laboratories charged with understanding and modeling the risks, human behaviors and motivations, and attacks within the context of Army cyber-operations. Led by Penn State University, the CSec CRA consists of over 40 researchers from the Army Research Laboratory, Carnegie Mellon University, Indiana University, the University of California Riverside, and the University of California Davis. The goal of the CSec CRA is to develop a *science of cyber-decision making* that enables actors in military environments to: *(i)* detect and characterize the risks and attacks present in the environment; *(ii)* understand and predict the motivations and actions of users, defenders, and attackers; *(iii)* alter the environment to achieve operation success. Practically speaking, we will enable defenders to answer, "Given a security and environmental state, what cyber-maneuvers best mitigate attacker actions and enable operation success?" Note that this is not a discrete and momentary analysis, but one that is continuous and adaptive within evolving state awareness.

The CSec CRA is broadly split into three research areas: detection, risk, and agility. The science of *detection* explores approaches that determine the current state of the environment, including identification of threats and on-going attacks. The science of *risk* is concerned with understanding and quantifying risk and its impact on the system. The science of agility–the focus of this paper–involves understanding *how* to respond to attacks, *what* maneuvers are available, and *when* those maneuvers should be executed to improve the security of the system.

2. AGILITY CHALLENGES

There are many ways to formulate and execute a MTD response to an environmental threat. In practice, the choice of MTD (or set of MTDs) to employ depends on a myriad of factors including the environmental state, resources at risk, threat model, and broader policy goals of the defender. For instance, maintaining the availability of a networked service in an Internet-facing DMZ requires different MTD maneuvers than protecting a backend database's integrity.

As a consequence, moving target defenses often involve complex tradeoffs. These tradeoffs have impacts on performance, availability and security, and introduce design and deployment challenges. In the CRA, we have identified three key challenges:

- *Concealing the strategy from the adversary* - The details of a maneuver must be known to friendly parties (where needed) and unknown to adversaries. For instance, when the defender migrates a service to hide its location from attackers legitimate users still need be able to access the service. Such maneuvers add overhead and complexity to the system.
- *Sustaining security across layers* - MTD must consider impacts at multiple layers of the system. For example, reconfiguration in the network layer may actually expose vulnerabilities in the software layer, e.g., expose open services. Axiomatically, the environment should always be more secure after a maneuver. As such, cross-layer MTD should be carefully designed to limit the scope of any maneuver to include only the elements that are necessary to achieve the security goals.
- *Managing cost* - MTDs often induce additional costs on the systems and users they defend. Maneuvers will typically introduce these overheads both during their execution and afterward. One has to take care to make sure that these costs are reasonable and do not unduly impact system operations.

The next three subsections consider each of these challenges in detail and posit ways of addressing them. Section 2.4 proposes a general model based on game theory that provides a platform for reasoning about MTD strategies.

2.1 Concealing the MTD Strategy

The effectiveness of many MTD strategies is based on their ability to conceal their activities and details from the adversary. Thus, any approach must be (*a*) unpredictable in its use, (*b*) varying widely enough to prevent the adversary from successfully guessing its activities, and (*c*) transparent to the adversary in its execution.

The other side of the equation is ensuring the resources of the environment remain known and available to the users. Securely communicating defensive maneuvers to legitimate parties is essential to any MTD framework. Ideally, an adversary should not be able to eavesdrop or spoof information about defense maneuvers. More realistically, the threat model must assume that an adversary can compromise some percentage of nodes in the network. The challenge is to find ways to limit the information that is exposed to each node in the network, allowing us to limit the impact of a compromise to a localized area. If the reconfiguration rules/strategies are leaked to an adversary, the adversary will not only regain the upper hand, but will potentially have a new advantage.

In some cases, a trusted public key infrastructure (PKI) can help secure network communication from eavesdropping; however, such schemes do not provide adequate protection against insider threats. Consequently, the MTD framework needs mechanisms to limit the information that is exposed to each node. For example, a controller that performs network reconfigurations might only provide each node with a sub-set of the reconfigurations; further, instead of using a shared group key to announce reconfigurations, the controller could use host specific keys. Under this scheme, if a node is compromised the adversary is privy only to the information that is exposed to that node. Similarly, operating system reconfigurations are local to the host. That is, they rarely need to be exposed to other entities in the network. Inexorably, localization strategies introduce complexity into the operation that needs to be balanced with the other costs.

A trusted certification authority (CA) may not always be available, e.g., in the case of mobile wireless networks. Randomized key predistribution schemes [4,11] are useful, but insufficient; current techniques do not provide a rich set of options for reconfiguration (e.g., k connectivity between nodes). One has to also make sure that any information that is exchanged for maneuver purposes are not leaked to a third party node on the open wireless medium.

2.2 Sustaining Security Across Layers

MTD mechanisms should observe the Hippocratic Oath: "*first do not harm*". Practically speaking, this means don't make the security posture of the environment worse. However, the complex interactions induced by MTD systems between systems and layers can inadvertently produce exactly this result.

Networked systems typically exhibit a large set of dependencies between the layers, spanning from the application all the way down to the physical layer. These dependencies will have to be accounted for and leveraged while designing a MTD framework. One can envision the interactions between the attacker and defender as a game; however, constructing games at a single layer can potentially result in an insecure system.

To illustrate, consider a wireless multi-hop network where a service is to be protected. The goal of the attacker is to locate the service, and disrupt its availability (e.g., by launching a DoS attack). The defender seeks to place the service at a location such that it can be reached by the other nodes in the network with least cost (e.g., the total distance to these nodes is minimized); however, knowing this strategy makes it easy for the attacker to target the service. In the simplest case, one could formulate the problem as a game wherein the defender tries to maximize a payoff function; maximizing the payoff would entail minimizing the distance to the other nodes from the service while at the same time, maximizing the distance from the adversary. The attacker tries to minimize the same payoff. One could come up with optimal mixed strategies for the attacker and defender in this case.

However, this representation of the service availability problem ignores the cross layer dependencies that arise during the migration process. For instance, in order to move the service from one node to another, the nodes will need to exchange information, e.g., establish TCP handshakes. In the process, even if the payload is encrypted, it is possible that the attacker is able to intercept exposed headers. This header information may allow the adversary to locate the service. Further, even if the headers were somehow concealed, the traffic access patterns may give the location away.

Consequently, agility must employ a combination of different

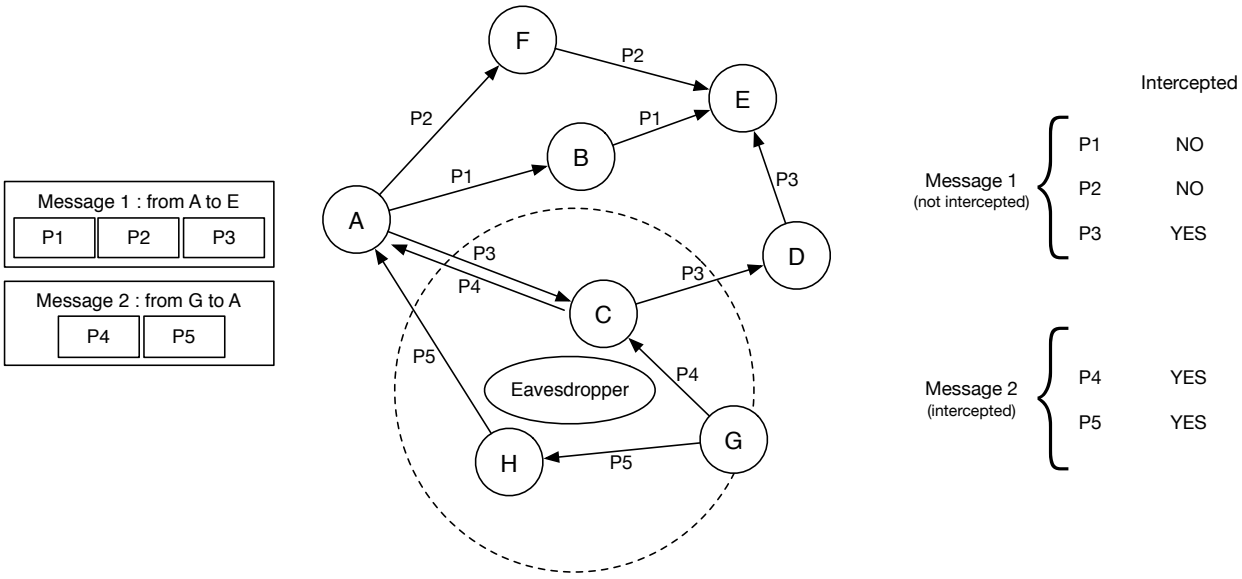

Figure 1: Multipath routing, coupled with power management, can prevent an adversary from intercepting all of the communication between two peers. Let's first consider the message sent from node A to node E. It is divided into three parts and each of them is sent to node E using a different path. If the eavesdropper is trying to intercept communications between A and E, he can only intercept the third part of the message while it is routed to and from node C. He is not able to understand the message. However, let's now consider the message sent from node G to node A. The first part is routed to node A through node C and the second through node H. Because the eavesdropper is in transmission range of both node C and node H, he will be able to intercept both parts of the message and therefore understand the message.

techniques across multiple layers. At the network layer, one can envision using multi-path routing (e.g., [3]) to dynamically vary the path of information in order to minimize the likelihood of packet interception. Multi-path routing can be extended in several ways to provide stronger notions of anonymity [2,10] to make it more difficult for an attacker to determine the source, destination, and event existence of control messages that guide reconfiguration. At the physical layer, features such as MIMO or power control can help limit the range of transmissions, making it more challenging for an adversary to eavesdrop. Figure 1 illustrates a simple scenario. In this scenario, the wireless network is made of several nodes. Two messages are being sent in multiple parts using multi-path routing. If we consider an eavesdropper trying to intercept these communications, the limited range of transmissions and the configuration of routes used by message parts will only allow him to fully intercept and understand one of the two messages.

However, these strategies impact network performance; use of longer (and possibly less reliable) routes could lead to higher load and delays. Similarly, topology control via either reducing power, or by using MIMO for creating null spaces would reduce both reliability of transmissions and cause paths to become longer. One can inject fake traffic to obfuscate the access patterns to the service under discussion but again this would result in an exacerbation of congestion within the network.

While the above discussion considered a wireless scenario, similar cases can be constructed for wired networks. Software defined networking (SDN) platforms combined with network function virtualization allow for the reconfiguration of network topology on the fly. Using these techniques, one could envision separating the identity of a node from its routing attribute (IP address)—though this would require dynamic reconfigurations of the controllers as well as routing tables at different nodes. Again, to hide the location of services one may consider the injection of fake traffic, the deployment of honeypots, or both. Some of the issues that arise in the wireless setting (e.g., eavesdropping on the open medium) do not exist here, but the scale of operations would make dynamic changes much more complex.

If we need only consider the connection of services to the network, cloud computing provides a natural platform for implementing agile services, as it creates a layer of indirection between clients (benign and malicious) and services and enables transparent migration of services. For example, different clients can be provided with different IP addresses and ports for the same service, enabling the agility framework to determine how to manage migration and replication for agility purposes. Thus, we can decide on a per-service basis whether to grant clients access to the "real" service (i.e., vs. a temporary front-end), whether and when to migrate the service, remedial actions to take on the service (e.g., to restore the service to a known-safe state), and how to divide clients among replicas of the service, and how to choose coordination among the replicas.

2.3 Managing Cost

Underlying our previous discussion is the inherent cost associated with any agility maneuver. These costs are reflected as either a loss in performance (either during the reconfiguration or in the final configuration), or as the need for additional assets. As a very simple example, if we consider service migration as a plausible option towards ensuring high availability, we must account for several costs. First, the migration strategy (where to migrate) and the migration trigger must be distributed to all affected parties. Second, during the migration, network capacity is used to move the process. Third, during migration, there may be interruptions in service—interrupts may or may not be acceptable depending on the situation.

Alternatively, we can use replication (instead of migration) to ensure availability. However, the more replicas, the higher the likelihood that an adversary will be able to correctly identify one and, depending on the threat, this may not be acceptable. Additionally, each replica incurs costs in terms of computing and storage as well

as the complexity of disseminating the location of the replicas to trusted parties. In some scenarios, a combination of migration and replication might be most successful, but even then, the right combination is hard to determine.

Even for internal reconfigurations, wherein the changes stay within the host, there are still overheads in terms of service availability. We discuss software agility further in the next section.

The costs of agility need to be managed. In other words, agility should not cause the system to be overwhelmed. One option is to the scope of the agility maneuver to reduce costs at fine time scales but only undertake larger scale changes at coarse time scales. Host level reconfigurations (e.g., ASLR to randomize the memory layout, software agility etc.) could drastically reduce the likelihood of certain types of threats. It would then become only necessary to determine what additional knobs will need to be tuned in order to cope with other threats.

The environment and applications should also determine the level of agility that is needed. In other words, given certain applications and settings, one could reduce the costs in providing agility. For example, in our earlier work, we consider selective encryption of video flows in order to provide protection against eavesdroppers [9]. Encrypting large video files for transfer over wireless settings could incur high energy costs and induce additional processing delays. If instead, one were to only encrypt parts of the video file, it may suffice in terms of obfuscating its content from an eavesdropper. One may also dynamically tune the level of encryption based on the importance of the content, the error rates on the network (which may help in protecting against an eavesdropper) and the energy/delay budget for the application.

2.4 Modeling MTD Games

MTD mechanisms must be based on a fundamental understanding developed via modeling attacker and defender interactions. As alluded to earlier, game theoretic models can inform interactions between the attacker and the defender. This can in turn help in providing valuable insights that will drive the design of algorithms for dynamically changing configurations.

Tunable hierarchical games. To cope with multi-scale, multi-objective adversarial behavior, one could consider what we call tunable hierarchical games. The basis for such games is as follows. The output of a game at one level will determine the level or risk associated with a game at a different level. Again, consider a network-level game, where the objective is to thwart DoS attacks (e.g., see prior work in [5,6], where backpressure is used to reduce the threat of DoS attacks in wireless multi-hop networks). In such scenarios, the solution (which comes at the expense of other performance penalties), is known to reduce the potency of TCP SYN flood attacks. Once a node is brought down via DoS, an adversary could effectively masquerade as that node to launch other potent (e.g., insider) attacks on other nodes. But once DoS attack likelihoods are reduced, the other attacks are likely to decrease as well.

Thus, at this point we could reduce the frequency with which a service is migrated from one node to another, i.e., the likelihood of potent attacks on the service is now decreased. Ultimately, this could influence the level of obfuscation needed at the link level (e.g., using friendly jamming). Naturally, the output of one game (e.g., using backpressure to determine the optimal rate at which TCP SYNs should be received at a given server node) influences the parameters of another game (e.g., service migration decision). We call these tunable hierarchical games. As with the previously proposed game models, the hierarchical games can be progressively refined based on knowledge gained regarding the attacker.

Imposing known signatures on legitimate functions. Instead of allowing all possible changes to network/system configurations, one could impose signatures to shape functions within the MTD framework. For example, access to certain resources (e.g., memory locations, paths) could be limited to certain times. Some nodes may be precluded from communicating or accepting new connections at some times. Certain OSes may be disabled at some times, while others are switched on. These patterns if unknown to the adversary, can help not only regain the first mover advantage, but also provide a quick way of detecting (failed) attack attempts. The challenge here is to determine signatures that are easy to implement and manage, and yet, hard for an adversary to guess. We expect that the these models could yield insights on the development of such signatures.

3. AGILITY MECHANISMS

Current systems are poorly equipped to prevent and withstand attacks, because defense decisions (if any) are hardcoded into system design and implementation. As a result, agility maneuvers are necessarily *reactive* and changing system behavior (e.g., writing and applying a security patch) takes time—time when the target services are vulnerable or unavailable. A more complete approach would include *proactive* maneuvers, such as MTD, to dynamically change the attack surface and reduce the time window for an adversary [7].

Adding MTD support requires developers to design and implement *agility mechanisms* to support both reactive and proactive maneuvers; broadly, they must satisfy the following requirements:

- Agility mechanisms should be flexible, allowing for a range of program changes. Such flexibility is challenging to obtain. While changing function bodies is often straightforward, changing data and function types is difficult and error-prone.
- The mechanisms must preserve application invariants and security policies, both during and after maneuvers.
- Mechanisms have to come at a reasonable cost. In general, support for maneuvers tends to impose memory, performance, and energy overheads. The value of a maneuver, and thus of the mechanisms that enable it, is a function of the effectiveness of the maneuver and the execution cost.

We propose several concrete mechanisms for agility: *system wrappers* which allow system design, implementation, and interaction with the environment to be modified at runtime; *capabilities*, a novel way of structuring system functionality which allows functional components to be turned on or off as needed; *distributors*, which allow services to be flexibly moved, split, and consolidated, so as to decouple task completion from where the service is executed.

3.1 Wrappers

Wrappers form a flexible reconfiguration layer that "wraps" entities such as software or services to enable agility maneuvers. We consider three general types of wrappers: communication, detection, and rewriting wrappers.

Communication wrappers intercept and rewrite the communication between an entity and its environment. For example, exception handlers can be intercepted and rewritten to allow fault recovery and self-healing. Another example would be a firewall-style wrapper that rewrites program input, e.g., SQL or HTML input sanitization to prevent SQL-injection or cross-site scripting attacks.

Detection wrappers permit placing (and removing) detection points, in accordance with requests from detection tasks. For example, system call arguments and return values can be intercepted to detect malicious processes. Another example is adding a monitor for application/system logs to detect anomalies.

Rewriting wrappers support nearly arbitrary software reconfiguration — changing programs statically or dynamically for MTD purposes. For example, switching encryption algorithms on-the-fly to offer improved confidentiality, or adding an integrity layer to servers. For example, adding integrity to Memcached to secure the communication with its peers; or applying patches to seal off buggy components in smartphone apps to preserve availability; or changing binaries to ensure binary variance as protection against attacks. In the same category, we can foresee changing the software's runtime environment to support MTD. For example, changing from unencrypted in-memory data structures to encrypted ones (followed by wiping the unencrypted storage) to preclude loss of confidentiality in the face to an attack.

The science of wrappers. Materializing our vision for wrappers will require several fundamental advances: ensuring the integrity of the wrapping process (wrappers cannot be taken over and subverted); ensuring the semantic integrity of code changes: change code statically or dynamically while providing certain security and safety guarantees. Modifying programs, their environments, or their executions, at runtime without impacting program semantics (or affecting only irrelevant parts of semantics, orthogonal to security) is difficult to establish as correct. This will require the development of a new theory (likely similar to noninterference) [12], which can be generalized to other security applications, e.g., deceiving adversaries without confusing legitimate users.

3.2 Capabilities

We propose the concept of *capabilities* as software units that can be added or deleted as needed. For example, in Web servers, capabilities include executing scripts or listing directories. On smartphones, communication (via Wi-Fi or 4G), or uploading files to a server, recording video, or point-to-point navigation are potential capabilities. Depending on system needs, we may need to remove existing capabilities, e.g., remove directory listing in Apache as protection against an exploit, or remove communication abilities from a smartphone when the smartphone enters hostile territory. Capabilities vary in extent and cut through program modules/methods/functions.

The science of capabilities. We plan to construct a theory of expressing the extent of a capability, as well as wrapper support for adding or removing capabilities on-the-fly. Defining formally, and automatically identifying, capabilities, as well as adding and removing capabilities on-the-fly will require advances in formal program modeling, analysis, reconfiguration, and security.

3.3 Distributors

Distributors support principled *service migration, consolidation, and redistribution*, allowing a single service to be performed on, or distributed to, multiple systems. For example, splitting a Memcached key-value server into several servers, to reduce processing delays and improve availability. Conversely, allowing distributed services to be performed on/distributed to one system; for example, a server S1 can take over the key-value services from server S2 (that is under attack) and service both S1 and S2's clients transparently to preserve availability.

Deception. Wrappers and distributors are ideal mechanisms for introducing deception: creating honeypots, fake copies of processes/services to observe how the attackers behave [8], and then feeding the adversaries distorted information to increase the adversary's cost and decrease the adversary's payout. Deception must be introduced carefully, to avoid confusing legitimate applications (which might not be aware of its existence).

A principled way of specifying correct, secure operation (e.g., via invariants), will have to be developed, as well as actions meant to bring software back to correct and secure operation by restoring programs and their execution to states and operations where they fulfill the invariants (at least partially). Establishing that, during repair and after repair, the program still abides by certain security policies is a challenge we plan to address explicitly.

The science of service migration and redistribution. We expect several challenges and scientific contributions: how to ensure that service migration (e.g., shipping existing key-value pairs) does not siphon off data? Or how to ensure that operations at the new server are semantically equivalent to the operations at the old server?

4. SELF-HEALING SOFTWARE

We now provide an example of MTD as an illustration of using agility maneuvers to cope with adversity, in this case smartphone software faults. Smartphone applications are prone to several classes of faults, including: unhandled exceptions, which will cause the OS to terminate the application; application crashes, e.g., due to semantic errors which will again lead to termination; permission violations, e.g., when the application attempts to access resources it does not have permission to; and so on. While the OS can restart the application when a fault occurs, application state can be lost, and the application will crash again when faulty code is invoked. To address this problem, in prior work [1] we added failure detection and recovery to Android applications by detecting faults and "sealing off" the faulty part of the application to avoid future faults. An overview of our approach is shown in Figure 2; the reader can ignore the blue callout boxes for now.

First, we extract an application model via static analysis; the model consists of high-level application states and their transitions. Next, to detect faults, we monitor application execution (dynamic analysis); when a fault is detected, we trace it to the method that has caused it and map the faulty state onto the model to find a safe state that precedes it. Then, we use binary rewriting to construct a "seal-off" patch that adds fault-handling code around the faulty method. Finally, the application is restarted and future faults in that method will be dealt with gracefully: intercepted and followed by a restart into the safe state, rather than terminated.

In sum, when using our implementation, applications can resume operation instead of repeatedly crashing, though with limited functionality if the fault is persistent. Our approach does not require access to application source code or any system (e.g., kernel-level) modification. We have demonstrated the utility of this approach on real faults in several widely-used, sizable Android applications: Facebook Mobile, NPR News, K-9 Mail, SoundCloud, APV PDF Viewer. Our approach managed to quickly recover from faults: bytecode rewriting took at most 22 seconds, while resuming the application in a safe state took at most 9 seconds.

4.1 From Smartphone Self-healing to General Agility

Self-healing shows how to cope with adversity on a device operating in a sensor-driven platform, unstable environments, with limited power and communication capabilities. We now discuss how this preliminary implementation encompasses many of the aspects needed for agility in general. Returning to Figure 2, note how our approach naturally lends itself to a broader, more general approach to agility as a means to implement MTD.

First, our dynamic analysis is based on monitoring application execution at fixed points, in this case Android's Dalvik VM log. These points can naturally be added/deleted as required; for example,

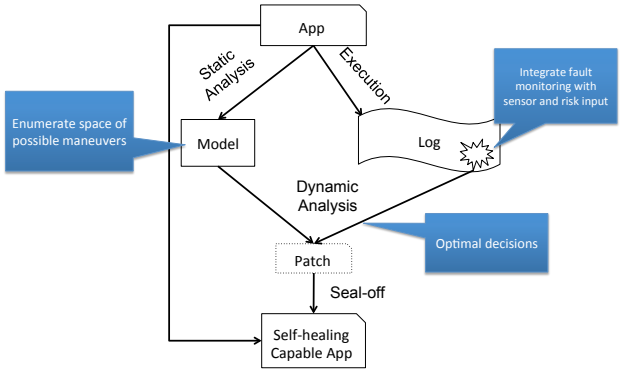

Figure 2: Smartphone self-healing: overview and integration with other components.

we could also monitor application system calls, application traffic, system parameters, etc., to detect a potential attack.

Second, our fault detection is based on a binary decision: if the fault manifests itself, perform self-healing. However, in some cases the decision to heal will depend on a cost-payoff analysis.

Third, our maneuver was simple and hard-coded: construct a patch and drive the application to a safe state. Agility allows for several maneuvers, e.g., no patching, just restart; or wipe the memory and the application space (in cases when protecting application secrets is more important than keeping the application functional); or restart the application and filter its incoming and outgoing traffic, etc. Deciding which maneuver makes sense at the current time and system state is another cost-payoff optimization problem.

Next, we provide a more detailed example of how the self-healing capability is factored into the cost vs. payoff analysis for agility in a bandwidth- and power-limited environment.

4.2 Agility Cost vs. Payoff

One way to compare the impact of agility maneuvers is to measure their impact on software capabilities and their reduction/restoration of some security level. For instance, we can consider the capability costs of sealing off a function, self-healing a function, and patching (if a patch exists). Both sealing off and self-healing can affect the capability and security of an application while patching (if a patch exists) would essentially restore both to full values. This relationship could be expressed for some arbitrary capability C_n as:

$$C_n(sealing\,off) \leq C_n(self\,healing) \leq C_n(patching)$$

and the level of security may be adjusted as S_n:

$$S_n(self\,healing) \leq S_n(sealing\,off) \leq S_n(patching)$$

Additionally, in constrained networks we can compare the costs of patching, sealing-off, and healing as a function of the power utilized to accomplish the agility maneuver. For instance, the three maneuvers require some computational power to apply; this can represent the cost value for the maneuver (e.g., $p(self\,heal)$, $p(seal\,off)$ and the $p(apply\,patch)$ where p represents power). The patching maneuver requires an additional cost: the power necessary to transmit the patch to the node. This will be at some cost to the nodes along the delivery path. This power consumption is a function of the size of the patch (bytes), the throughput for each individual link (bytes/second) and the power consumption rates for nodes to transmit and receive data (usually expressed in watts/hour).

As an example, let us assume all nodes have equal power consumption communication rates and the number of intermediate nodes between the node possessing a patch and the node requiring the patch is $n-1$. Given a patch size of b bytes, throughput between arbitrary nodes i and j is $t_{i,j}$ and the power to transmit is p_{tx} and power to receive p_{rx}, and the cost to deliver the patch is

$$P_{patch\,delivery} = bt_{0,1}p_{tx} + \Big(\sum_{i=1}^{n-1} bt_{i,j}(p_{tx}+p_{rx})\Big) + bt_{n-1,n}p_{rx} \quad (1)$$

Overall the cost to patch is represented as:

$$p_{patch} = p_{apply\,patch} + p_{patch\,delivery}$$

In situations where the network is constrained, the distance from the patch location to the node is many hops (large n) and/or the patch size is large. We may find that $p_{patch} >> p(self\,heal)$ and $p_{patch} >> p(seal\,off)$.

5. CONCLUSIONS

This paper has considered a new science of environment reconfiguration called system agility. The promise of such a science is that it will provide a generalized means of continuously altering the environment to address evolving needs. Yet, agile systems face complex challenges in realizing this vision. Systems such as those employing Moving Target Defenses must evolve in unpredictable ways that potentially lead to decreased reliability and suboptimal performance. In the worst case, these systems can introduce new vulnerabilities and decrease the security of the entire system.

Consisting of academic, industrial, and government partners, the Cyber-Security Collaborative Research Alliance has launched a 10 year effort to begin to develop this new science. It is through these efforts that we hope to help transform the networks of today from the static target-rich environments that exist today to the defensive and nimble that they need to be.

Acknowledgements. This research was sponsored by the Army Research Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied of the Army Research Laboratory or the U.S. Government.

6. REFERENCES

[1] T. Azim, I. Neamtiu, and L. Marvel. Towards self-healing smartphone software via automated patching. In *29th IEEE/ACM International Conference on Automated Software Engineering (New ideas track), ASE 2014*, September 2014.

[2] H. Choi, P. McDaniel, and T. L. Porta. Privacy Preserving Communication in MANETs. *Proc. of IEEE SECON*, 2007.

[3] J. Eriksson, M. Faloutsos, and S. V. Krishnamurthy. Routing amid colluding attackers. In *In ICNP '07*, 2007.

[4] L. Eschenauer and V. Gligor. A key management scheme for distributed sensor networks. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, 2002.

[5] V. Gupta, S. Krishnamurthy, and M. Faloutsos. Denial of service attacks at the mac layer in wireless ad hoc networks. In *MILCOM 2002. Proceedings*, volume 2, pages 1118–1123. IEEE, 2002.

[6] V. Gupta, S. V. Krishnamurthy, and M. Faloutsos. Improving the performance of tcp in the presence of interacting udp flows in ad hoc networks. In *NETWORKING 2004. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications*, pages 64–75. Springer, 2004.

[7] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang. *Moving Target Defense*. Springer, 2011.

[8] A. Juels and R. L. Rivest. Honeywords: making password-cracking detectable. In *ACM Conference on Computer and Communications Security'13*, pages 145–160, 2013.

[9] G. Papageorgiou, J. Gasparis, S. V. Krishnamurthy, R. Govindan, and T. La Porta. Resource thrifty secure mobile video transfers on open wifi networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13. ACM, 2013.

[10] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous Connections and Onion Routing. *Journal on Selected Areas in Communication Special Issue on Copyright and Privacy Protection*, 1998.

[11] P. Traynor, H. Choi, G. Cao, S. Zhu, and T. L. Porta. Establishing pair-wise keys in heterogeneous sensor networks. In *Proceedings of IEEE INFOCOM*, 2006.

[12] D. Von Oheimb. Information flow control revisited: Noninfluence= noninterference+ nonleakage. In *Computer Security–ESORICS 2004*, pages 225–243. Springer, 2004.