**Technical Report**
**1228**

# Survey of Cyber Moving Targets
# Second Edition

B.C. Ward
S.R. Gomez
R.W. Skowyra
D. Bigelow
J.N. Martin
J.W. Landry
H. Okhravi

17 January 2018

**Lincoln Laboratory**
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
*LEXINGTON, MASSACHUSETTS*



**This material is based upon work supported under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001.**

This report is the result of studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology. This material is based upon work supported under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Air Force.

Massachusetts Institute of Technology
Lincoln Laboratory

# Survey of Cyber Moving Targets
# Second Edition

*B.C. Ward*
*S.R. Gomez*
*R.W. Skowyra*
*D. Bigelow*
*J.N. Martin*
*J.W. Landry*
*H. Okhravi*

*Group 58*

Technical Report 1228

17 January 2018



Lexington Massachusetts

This page intentionally left blank.

## ABSTRACT

This survey provides an overview of different cyber moving-target techniques, their threat models, and their technical details. A cyber moving-target technique refers to any technique that attempts to defend a system and increase the complexity of cyber attacks by making the system less homogeneous, static, or deterministic. This survey describes the technical details of each technique, identifies the proper threat model associated with the technique, as well as its implementation and operational costs. Moreover, this survey describes the weaknesses of each technique based on the current proposed attacks and bypassing exploits, and provides possible directions for future research in that area.

This page intentionally left blank.

## ACKNOWLEDGMENTS

The authors would like to sincerely thank the contributors to the first edition of this survey: Mark Rabe, Travis Mayberry, William Leonard, Thomas Hobson, and William Streilein.

This page intentionally left blank.

# TABLE OF CONTENTS

## TABLE OF CONTENTS
## (Continued)

**Page**

## TABLE OF CONTENTS
## (Continued)

## TABLE OF CONTENTS
## (Continued)

**Page**

# 1. INTRODUCTION

This survey provides an overview of different cyber moving-target techniques, their threat models, and their technical details. A cyber moving-target technique refers to any technique that attempts to defend a system and increase the complexity of cyber attacks by making the system less homogeneous, static, or deterministic [1]. In this survey, we describe the technical details of each technique, identify the proper threat model associated with the technique, and identify its implementation and operational cost. Moreover, we describe the weaknesses of each technique based on the current proposed attacks and bypassing exploits, and provide possible directions for future research in that area.

## 1.1 TAXONOMY OF MOVING-TARGET TECHNIQUES

We identified five top-level categories and two subcategories of moving-target techniques. Figure 1 illustrates these categories. Here we give a short description for each category.

1. **Dynamic Data:** Techniques that change the format, syntax, encoding, or representation of application data dynamically.
2. **Dynamic Software:** Techniques that change an application's code dynamically. The change includes modifying the program instructions, their order, their grouping, and their format.
3. **Dynamic Runtime Environment:** Techniques that change the environment presented to an application by the operating system during execution dynamically.
   (a) **Address Space Randomization:** Techniques that change the layout of memory dynamically. This can include the location of program code, libraries, stack/heap, and individual functions.
   (b) **Instruction Set Randomization:** Techniques that change the interface presented to an application by the operating system dynamically [2]. The interface can include the processor and system calls used to manipulate the I/O devices.
4. **Dynamic Platforms:** Techniques that change platform properties (*e.g.*, CPU, OS) dynamically. This includes the OS version, CPU architecture, OS instance, platform data format, *etc.*
5. **Dynamic Networks:** Techniques that change network properties including protocols or addresses dynamically.

## 1.2 TAXONOMY OF ATTACK TECHNIQUES

The effect of each moving-target technique is described in terms of the attack technique that it mitigates. Here we provide a brief definition for the attack techniques used in this report. The

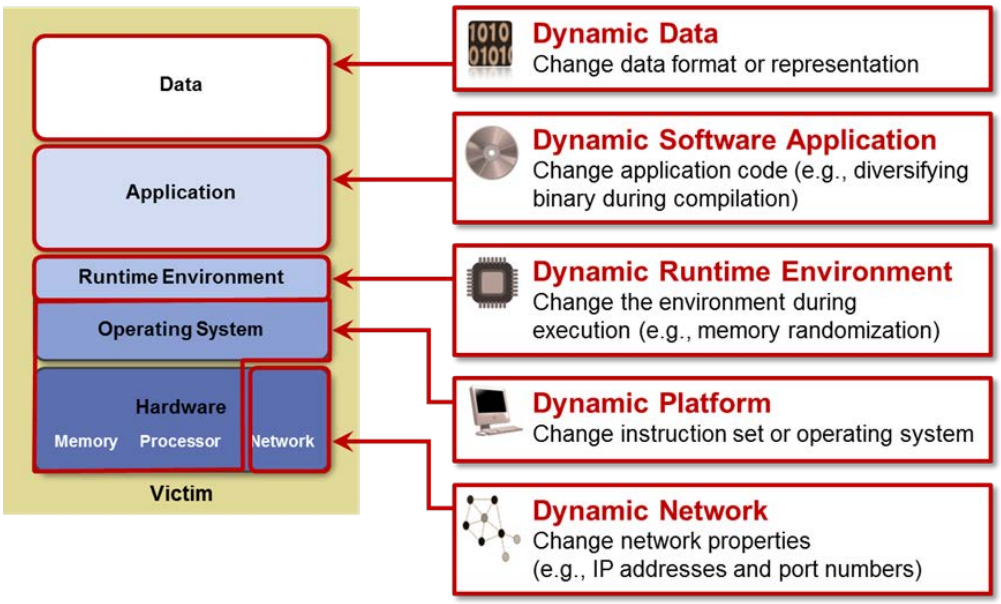


*Figure 1. Different categories of moving target techniques.*

taxonomy of attacks is a customized version of the Common Attack Pattern Enumeration and Classification (CAPEC) attack categories [3].

1. **Data Leakage Attacks:** Attacks that actively target important information on a system (*e.g.*, leakage of cryptographic keys from memory).
2. **Resource Attacks:** Attacks that exhaust or manipulate shared resources in a system (*e.g.*, denial-of-service using CPU saturation).
3. **Injection:** Attacks that force malicious behavior in the system.
   (a) **Code Injection:** Attacks that force malicious behavior in the system by inserting malicious code (*e.g.*, buffer overflow and script injection; SQL injection).
   (b) **Control Injection:** Attacks that force malicious behavior in the system by manipulating the control of the system and without malicious code. Control can include timing, ordering, and arguments of different operations. (*e.g.*, chaining existing code snippets together to achieve malicious behavior (return-oriented programming (ROP) [4]).
4. **Spoofing:** Attacks that fake identity of a user or a system (*e.g.*, man-in-the-middle attack; phishing attack).
5. **Exploitation of Authentication:** Attacks that compromise explicit or implicit authentication processes in a system (*e.g.*, cross-site scripting).
6. **Exploitation of Privilege/Trust:** Attacks that misuse granted privileges (*e.g.*, session hijacking).

7. **Scanning:** Attacks that collect information passively or non-intrusively (*e.g.*, port scanning).

8. **Supply Chain/Physical Attacks:** Attacks that target supply chain or physical security of a system (*e.g.*, malicious processor).

### 1.3 TAXONOMY OF ENTITIES PROTECTED

Each moving-target technique is designed to protect specific entities in a system. Here we provide a taxonomy of entities protected by the techniques we analyze in this survey.

1. **Applications:** All or specific applications are protected from network entities or other applications running on the same system (*e.g.*, protecting application memory location from other applications; protecting database applications).

2. **Operating System:** The operating system is protected from network entities or malicious applications running on top of it. This protection usually attempt to prevent privilege escalation or access to the kernel-space and other applications (*e.g.*, sandboxing suspicious applications).

3. **Machine:** All or specific types of machines (also called clients, hosts, or servers) are protected from other network entities (*e.g.*, changing the IP addresses to make scanning more difficult; protecting web servers behind a firewall).

4. **Network:** A network or subnet is protected from other networks (*e.g.*, dynamically changing IP address on the VPN gateway to protect against malicious connections).

5. **Traffic:** Confidentiality and/or integrity of all or specific types of network traffic is protected (*e.g.*, dynamically changing protocols to make traffic injection more difficult).

6. **Session:** A set of user operations (a session or a transaction) is protected from other untrusted operations (*e.g.*, a secure web transaction is protected from other web pages browsed on the same machine).

7. **Data:** Confidentiality or integrity of data handled by applications or stored on the machine is protected (*e.g.*, changing data encoding to prevent malicious data modifications).

### 1.4 CYBER KILL CHAIN

Each moving-target technique is focused on disrupting certain phases of a successful attack. For instance, while a technique may make it less likely for an exploit to succeed during launch, another focuses on making information collection on the target more challenging. In this survey, we try to identify the phase of an attack each technique is targeting. These phases are also referred to as the cyber kill chain.

1. **Reconnaissance:** The attacker collects useful information about the target.

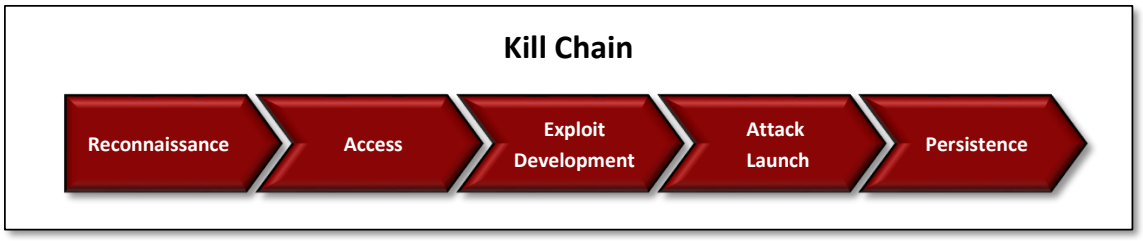


*Figure 2. Cyber kill chain.*

2. **Access:** The attacker tries to connect or communicate with the target to identify its properties (versions, vulnerabilities, configurations, *etc.*).
3. **Exploit Development:** The attacker develops an exploit for a vulnerability in the system in order to gain a foothold or escalate his privilege.
4. **Attack Launch:** The attacker delivers the exploit to the target. This can be through a network connection, using phishing-like attacks, or using a more sophisticated supply chain or gap jumping attack (*e.g.*, infected USB drive).
5. **Persistence:** The attacker installs additional backdoors or access channels to keep his persistence and access to the system.

Figure 2 illustrates the cyber kill chain used in this survey. We choose this kill chain because it is well suited for the types of protections offered in moving-target defenses. There are other types of kill chains proposed in the literature that are better suited for specific domains in cyber. They include cyber war kill chain (phases: reconnaissance, weaponize, delivery, exploit, install, command & control, and act on objectives), action-oriented kill chain (phases: deter, protect, detect, react, and survive), detection kill chain (phases: herd, perturb, disturb, *etc.*), and others.

### 1.5 TAXONOMY OF WEAKNESSES

When identifying the weaknesses associated with each moving-target technique, we consider four types of weaknesses that can make the technique ineffective. One or all of these weaknesses can exist in a technique and any of them can defeat the purpose of that technique.

1. **Overcome Movement:** With this weakness, the movement happens and the pattern of movement is random or controlled, but the adversary can still attack the surface protected by the MT technique. For example, injecting many copies of the exploit to overcome address space randomization is a form of overcoming the movement.
2. **Predict Movement:** With this weakness, the movement happens and the pattern of movement is random or controlled, but the adversary can still attack the surface protected by the MT technique. For example, leaking addresses to predict the location of libraries is a form of predicting the movement in address space randomization.

MT technique. For example, leaking addresses to predict the location of libraries is a form of predicting the movement in address space randomization.

3. **Limit Movement:** With this weakness, the movement happens, but the pattern of movement is limited by the adversary's actions. For example, the adversary can fill up memory to limit the randomness in address space randomization (a.k.a. code spraying).

4. **Disable Movement:** With this weakness, the adversary explicitly disables the movement. For example, address space randomization can be disabled in the OS by pushing a bad configuration.

### 1.6 SCOPE

This survey provides a complete representative set of moving target techniques from open and public sources of information. Specifically, the techniques described herein were identified through a comprehensive literature review. We manually reviewed the proceedings of the top six security conferences, ACM CCS, IEEE S&P, NDSS, USENIX Security, ACSAC, and RAID to identify all relevant moving-target techniques. Furthermore, we performed keyword searches of the ACM and IEEE databases, as well as more general scientific databases. Finally, we conducted Google searches to find any remaining papers, as well as commercial products leveraging moving-target defenses. All together, this search identified over 100 sources published after the previous version of this survey in 2011 [5]. From these sources, we added 52 new techniques to this document for this version. This document focuses specifically on techniques for cyber moving targets, so papers that did not present new techniques, but instead conducted analysis, meta-analysis, or evaluations of existing techniques, were not specifically incorporated into this document. Furthermore, there are several papers that present similar techniques, or extensions to previous techniques, and were therefore merged herein for clarity.

Although we expect that there are other commercial products or academic projects with different names that implement similar moving-target techniques or some combination thereof, to the best of our knowledge, they are not fundamentally different in their concepts and workings from the techniques presented herein.

### 1.7 ORGANIZATION

The rest of this document is organized according to the moving-target-techniques taxonomy described earlier. We summarize each technique, as well classify the technique against the taxonomies described above. We also present relevant details pertaining to the threat model the technique defends against; the entities that are protected by the technique; any details pertaining to how the technique can be deployed, including any interdependencies among different techniques; any overhead associated with the technique; the system components that must be modified to use the technique in production; the complexity of implementing and operating the technique; the phase(s) of the cyber kill chain that the technique helps mitigate; the weaknesses of the technique; the impact the technique has on the attacker; any opportunities that we identified as avenues for

further research; the availability of the technique; the funding source of the research that developed the technique; and any other miscellaneous considerations or notes that we found relevant to the technique. In addition, we highlight the costs to deploy each technique, the complexity of both implementing and operating each technique, and which aspect of the cyber kill chain the technique targets.

# 2. DYNAMIC DATA

## 2.1 DATA DIVERSITY THROUGH FAULT TOLERANCE

**Defense Category:** Dynamic Data

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource

**Details:** This technique [6] was not designed to fight malicious input directly but it is more focused on unintentional faults. Since it reprocesses data and votes on the results, it could help combat an attacker that is trying to manipulate or corrupt the output of a program or service.

**Description:**

**Details:** This technique aims to increase the fault tolerance of an application by reevaluating the input to a program using a different algorithm. These different algorithms can produce exact equivalent output or they could be general algorithms that produce approximations of the original output. The idea is a possible fault or corner-case for a specific input might be avoided if it is calculated in a slightly different or semantically equivalent fashion. The technique builds on the idea of $N$-version programming but uses a data-centric version of it the authors refer to as $N$-copy programming. Input is passed into independently developed versions of a program. The output of these is then passed to a voter that decides if the input is acceptable. If the output does not look acceptable, a new algorithm is chosen to process the input and the cycle is done again. If exact algorithms are being used, the voter can use the majority output as the good output. If it switches to more generic algorithms that produce approximations, then the voting can become subjective because the copies could produce different results that are still acceptable.

**Entities Protected:** This technique aims to protect a program by ensuring the output is acceptable.

**Deployment:** This would be implemented into the code of a program on a system.

**Execution Overhead:**

- There may be some additional processing overhead imposed if the program needs to reprocess the input
- Running multiple copies of a program and waiting for voting results will add additional overhead

**Memory Overhead:**

- Extra memory used by running multiple versions of the program (roughly $N$ times for $N$ copies)

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☒ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** The technique assumes that $N$ different (independent) versions of the algorithm can be built.

**Weaknesses:** This technique relies on voting so it is still possible for an attacker to corrupt all or the majority of the processes in order to bypass the added protection. It may also still be possible for an attacker to create output that still looks valid to the output checker so it is not rerun again with different algorithms. Another possibility is that the differing algorithms might have no effect on the malicious input.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** If an attacker is attempting to corrupt or manipulate the output of a program, this could make it more difficult. The technique is mainly focused on integrity protection. If the algorithms used to process the input are not sufficiently different between retries, an attacker may still be able to complete their objective. If an attacker can accomplish their goals without needing the programs to output, this may not have much of an impact.

**Availability:** This technique was tested by the authors but does not appear to be publicly available.

**Additional Considerations:** Creating a component that can accurately detect the validity of the output could be difficult for a program or service with varying and dynamic output. Also developing $N$ different algorithms is time consuming and requires redevelopment of an application.

**Proposed Research:** Some problems would need to be solved to make this technique more reasonable. One of those is coming up with a reliable way to determine if output of a program is valid. There are many applications and services now that have very dynamic and varying outputs so it may not be trivial to determine if output is valid. The same can be true for the varying input processing algorithms. It may not be an easy task to develop multiple ways to process the input inside of the application. It may also not be an option to use more approximate methods if the accuracy of the output is important. Automating the diversification of an algorithm is an important future direction.

**Funding:** NASA

## 2.2 REDUNDANT DATA DIVERSITY

**Defense Category:** Dynamic Data

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource and Code Injection

**Details:** This technique [7] aims to help mitigate attacks that target specific data inside of an application by way of malicious input. Each variant is running a different transformation of the data such that one input would not be able to change all variants. This would cause a divergence and it would be detected by the variant monitor. The change can also be done at a lower level separating the address space of each variant or running each variant with different instructions. This helps mitigate some code injection attacks or injection attacks that rely on specific memory addresses.

**Description:**

**Details:** This technique is a variation of the N-variant programming technique. It involves running multiple copies of a program that each run transformations of the original data being protected without having to rely on secrets. These transformations should be semantically equivalent and reversible. A monitor can watch the values of the data in each variant to detect if there is a divergence and take appropriate action. This can be implemented on different levels such as having variants use different memory address spaces, different instructions, or different data representations.

The specific method for this technique analyzed was using different data representations. This was implemented to protect user identification (UID) and group identification (GID) that are used for determining permissions. This is implemented into the system kernel, new system calls are created to allow for synchronization, and other system calls are modified accordingly to support the modified data. Each variant is modified to use new system calls for synchronization and to support the new UID and GID representations. The variants synchronize on system calls. Whenever one variant reaches a system call, it waits for the other to reach it as well. The inputs to the system calls are verified before execution. The system call is only executed once and the results are passed to each variant if it was an I/O-based system call. If the program uses external file, such as configuration files, a new one is created and tailored toward the specific variant. If the program used the password file on the system and it contained some of the data being randomized, a new password file would need to be created for each variant.

**Entities Protected:** This technique aims to protect data entities inside of a running program on systems.

**Deployment:** Depending on the types of data being protected, it could be deployed at different levels. The implementation described is implemented into the operating system kernel.

**Execution Overhead:**

- Running the Apache Web Server unsaturated with 2-Variant UID imposed a 13% throughput overhead and 14% latency overhead.
- Running the Apache Web Server saturated with 2-Variant UID imposed a 58% throughput overhead and 135% latency overhead.

**Memory Overhead:**

- There will be additional memory used by running multiple variants simultaneously

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** This should not be combined with diversity techniques that change the behavior of the program or make it perform semantically different. Such a technique would cause a divergence that would trigger a detection in their monitor.

**Weaknesses:** An attacker could still target data parts of an application that are not randomized if they can be used to mount an attack. An attacker could also try to use advanced control-injection attacks that could still potentially affect many or all variants. Also the technique proposed is very limited in scope (only a very small portion of data on the system is randomized).

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This could make it much more difficult for an attacker to corrupt certain important parts of an application if it was being protected properly by this technique.

**Availability:** This technique was prototyped by the authors but was not publicly released.

**Additional Considerations:** Techniques like this would have a larger overhead for computation-intensive programs. Each variant would have to do the expensive computations. In addition, it can be very challenging to expand this technique to the majority of data being processed on the system. There was no mention of recovery if the monitor detects something malicious.

**Proposed Research:** This technique was currently only implemented to protect data inside of the application. It could be extended to include some of the lower-level diversification techniques

also described in the paper such as instruction set tagging and address space separation. This would make the application more resistant to different code injection attacks but would also add additional execution overhead as well. Additional research may also be needed to overcome possible false positive detections due to accidental divergences. These could happen because of operating signals reaching variants in different positions of execution.

**Funding:** National Science Foundation

### 2.3 DATA RANDOMIZATION

**Defense Category:** Dynamic Data

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [8] helps protect against code-injection attacks by randomizing any code injected into the program. All data that is written to memory within a certain class is randomized with a random key. This also helps protect against attacks that target pointers in general such as function pointers or return addresses. These would also be randomized by this technique using different keys. In addition, this technique would also provide some protection from attacks that attempt to read or write arbitrary memory locations. Any functions that attempted to write something would have that randomized as it was put into memory and reading arbitrary memory locations would result in that data being randomized with the key.

**Description:**

**Details:** This is a compiler-based technique that provides probabilistic protection by randomizing all the data that it stores in memory. All operands in a program within a class that read and write memory are instrumented to perform an XOR of the data with a random key. All operands that reference the same objects are grouped together. Each of these groups is randomized with a different key that is generated when the program is started. These groups are found during compile-time by using static analysis within the compiler. To improve performance, operands that are classified as safe are not instrumented. An operand is considered safe if runtime access to that operand can never violate memory safety. The compiler will then insert instructions that perform the XOR operations for reading and writing to memory in the appropriate locations. This technique also supports libraries. Wrappers can be created for the library functions and system calls that receive or return pointers.

**Entities Protected:** This technique protects the data applications store in memory.

**Deployment:** This technique would be implemented in a compiler on a system. Each program that wanted to use this technique would need to be compiled with this new compiler.

**Execution Overhead:**

- The average overhead for the tested benchmarks was 11% but it can be a wide range in either direction.

**Memory Overhead:**

- The tested benchmarks had an average memory overhead of 1%.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☒ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** A good source of randomness for the key generation.

**Weaknesses:** It is still possible for an attacker to guess the randomization key to be able to read/write data to/from memory (technique assumes memory secrecy). It is also still possible to attempt to brute force the desired keys. This could result in a large number of program failures that would increase the probability of detection. An attacker may also be able to get to the desired memory object if there is a vulnerability in the same group of operands since they would use the same key. In order for this to be effective, it requires that all libraries also be protected via wrappers. If any libraries are overlooked, that opens the possibility to bypass this technique.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique would make exploiting a protected application more difficult. An attacker would have to find a method to either leak the keys or guess the keys used to randomize the data.

**Availability:** This technique was prototyped by the authors but was not publicly released.

**Additional Considerations:** It requires program recompilation. The size of the programs increased by averages between 15% and 30%. Applying this technique to a wide range of programs can make it impractical.

**Proposed Research:** One larger direction this technique could take would be to combine with other memory protection techniques such as Address Space Randomization. This would put further burden on the attacker that is trying to execute low-level attacks. Also it would be important to study what types of attacks can be mounted without crossing the groups (classes).

**Funding:** Microsoft Research

## 2.4 END-TO-END SOFTWARE DIVERSIFICATION

**Defense Category:** Dynamic Data, Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Exploitation of Authentication

**Details:** This technique [9] has the potential to defend against different levels of code injection as well as some authentication attacks. Randomizing the instruction sets, script Application Programming Interface (API) randomization, randomizing the reference names of stored data, and randomizing components of code can help fight high-level code injection attacks like Structured Query Language (SQL) injection attacks as well as low-level code injection attacks that target the internal application. They can also help fight attacks that compromise authentication like Cross-Site Scripting (XSS) attacks that might try to inject code at a high level. Other diversification methods that can be used with this technique can help mitigate injection at additional levels.

**Description:**

**Details:** The idea of this technique is to compose many different randomization methods and apply them to aspects of a service that does not affect the functionality of the program. This would involve building functionality into the core or subsystems of a service that allows various aspects to be randomized. The example in the paper is diversifying an Internet service. Some of the proposed diversification methods include changing Hypertext Transport Protocol (HTTP) keywords/syntax/headers/content encoding, Hypertext Markup Language (HTML) Document Object Model (DOM) structures/identifiers, Structured Query Language (SQL) keywords/syntax, database server instruction set/Internet Protocol address/port number/memory layout, database table names/column names, web server instruction set/memory layout, and local files used by the servers. There are other aspects of such a service that could also be diversified while not directly affecting the service functionality. Each method of diversification would have its own side effects and performance implications. There may also be other parts not identified that could also be used for diversification and coming up with a complete list is a difficult problem. The number and type of things that can be diversified will depend on the desired service and the software being used to provide that service. Another aspect of this technique is how often the randomization happens. In the case of a web service, it can be set up so that each user instance has a different randomization plan. It could also be implemented that each user request causes a new randomization of some of the methods.

**Entities Protected:** This technique aims to protect a web server.

**Deployment:** This would be deployed on a server.

**Execution Overhead:**

- This will vary with the number and type of diversification techniques implemented. Low-level emulated instruction randomization techniques would have a much higher overhead than randomizing the table names in a database. The overhead may be significant.

**Memory Overhead:**

- This will also depend on the diversification techniques implemented. If memory layout randomization is enabled, this could impose some overhead depending how it is implemented.

**Network Overhead:**

- Depending on the transformations applied to network protocols, this could increase the size of network traffic or increase the processing time of the traffic.

**Hardware Cost:**

- None

**Modification Costs:**

☒ Data

☒ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** Not all combinations may be desirable. Combining methods of randomization could affect the application in undesirable or unexpected ways. Certain combinations could also result in a large overhead.

**Weaknesses:** Weaknesses associated with this technique will vary depending on the randomization techniques implemented. Each technique will have its own weaknesses associated with it and combining techniques could introduce additional weaknesses not present in the independent methods. Some randomization methods may be limited by factors in the system such as the architecture. Despite all randomization, a higher level protocol may be vulnerable to attacks.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☒ Limit Movement

☒ Disable Movement

**Impact on Attackers:** If that attacker can leverage vulnerabilities in a service that allow him or her to take control of the flow of the program, he or she could still leverage more advanced techniques that do not rely on code injection. Implementing many of these methods will increase the amount of work an attacker has to do to exploit the system.

**Availability:** This was a research idea by the authors and did not appear to be implemented or made available

**Additional Considerations:** The paper lacks many specifics. It is only applied to a web server. The actual impact of randomization in unknown. The overhead can be very large. Modifying the code to support all these additional randomization abilities could introduce additional

bugs/vulnerabilities. Modifying the code to support all these additional randomization abilities could increase the maintenance complexity of application. Determining which components and subcomponents of an application or service could be a difficult and time-consuming task. The proposed randomization methods do not fix security vulnerabilities or other logic errors that are part of the design of the software. A similar technique is proposed in [10].

**Proposed Research:** This technique proposes many possible ways a specific web service could be randomized. Coming up with a method to identify and test these methods is not an easy task. Any part that is overlooked could become a potential attack vector. Determining which of these methods can be safely combined could also be a difficult task. It could be the case that combining two methods result in something breaking elsewhere in the service or system. Certain techniques will also have varying impacts on performance and the combination of different methods could cause unexpected performance issues. Overall, the composition of different randomization and diversification methods would need to be further researched for this technique to be more feasible.

**Funding:** Unknown

### 2.5 DIGLOSSIA

**Defense Category:** Dynamic Data

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** SQL Command Injection

**Details:** This technique [11] is designed to detect SQL and NoSQL command injection via user-provided input. The basic technique is extendable to other formats vulnerable to this type of command injection.

**Description:**

**Details:** This technique detects SQL and NoSQL command injection by dynamically remapping all SQL/NoSQL queries to shadow character sets immediately before query processing, preserving all user-provided input in its original character set. This serves as a type of lightweight taint tracking, and enables a comparison between the original query and the final user-tainted query using a technique called *dual parsing*. If the dual results of such parsing are not "syntactically isomorphic" or if any command sequences are recognized as being user-provided by virtue of appearing in the original character set rather than the shadow character set, command injection is declared and appropriate action can be taken. This technique is designed to be complementary to SQL/NoSQL input sanitization and serves as a backstop to those techniques at the interpreter level if the original program code does not follow best practices.

**Entities Protected:** Applications running in interpreted environments that take user-provided inputs as parameters to SQL or NoSQL queries.

**Deployment:** This technique was prototyped as a modification to the PHP interpreter and invoked by webserver requests. It could presumably be implemented in other runtime interpreters for languages other than PHP.

**Execution Overhead:**

- Maximum of 13% overhead in testing; on the order of 1-2 milliseconds.

**Memory Overhead:**

- Not specifically measured or discussed, but likely negligible.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** None, but could conflict with other language-interpreter modifications

**Weaknesses:** This technique cannot be used where commands are expected to be part of the provided input. Although acting on user-provided commands is typically regarded as poor programming practice, it has use in certain application areas and is found in the wild. Because the defense is implemented at the program-interpreter level, such programs could not be used on the modified interpreter, or that interpreter would have to support the selective disabling of the defense. The defense must be implemented per-interpreter or per-runtime environment. The technique cannot detect the use of integer literals as command parameters, considered by some to be a form of command injection. If input-tainted strings are passed to entities outside the environment, the tracking becomes imprecise and may allow attacks to pass undetected.

**Types of Weaknesses:**

- ☐ Overcome Movement
- ☐ Predict Movement
- ☒ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** Attackers are unable to perform SQL/NoSQL command injection on applications protected by this technique, provided that the queries are not partially handled by outside entities.

**Availability:** System is not publicly available.

**Additional Considerations:** The performance overhead for this technique is acceptable for network applications where a 1-2 millisecond delay is easily lost in overall network timing, but would quickly become overwhelming in a locally run query-intensive application.

**Proposed Research:** SQL command injection is a well-studied area and the need for input sanitization is well-known. Implemented in the interpreter, this technique serves as a backstop for coding best practices in sanitization. Further research to generalize the technique to a broader range of environments would help make it more widely applicable.

**Funding:** National Science Foundation

### 2.6 NOMAD

**Defense Category:** Dynamic Data

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Automated Web Bots

**Details:** This technique [12] is designed to thwart web bots that automate certain tasks and activities, such as account registration and forum posting, that are properly intended to be performed by humans. It targets simple bots relying on HTML metadata by randomizing HTML elements without hindering normal human activities.

**Description:**

**Details:** This technique prevents simple web bots from performing certain classes of undesirable activity, such as spamming comment forms or automatic web-based games, by randomizing HTML element names in served web pages. Because humans interact only with rendered content while web bots often rely on the metadata encoded within raw HTML, there should be no impact to human use of such sites while web bots become confused and are unable to perform their intended task. The HTML element names rotate on a regular basis in order to prevent a bot from "learning" the updated names of the elements. This technique is implemented either as a web server component or as an independent stand-alone proxy that dynamically rewrites HTML as data is passed through. Neither the primary content nor the Document Object Model of the page are modified.

**Entities Protected:** Web forms that are not intended to be automated.

**Deployment:** This technique can be deployed as a modification to webservers, or as a web proxy.

**Execution Overhead:**

- Average overhead of approximately 14%; maximum overhead around 30%

**Memory Overhead:**

- Not reported, but not expected to be significant

**Network Overhead:**

- Average webpage size increases by around 100 bytes in experimentation.

**Hardware Cost:**

- None

**Modification Costs:**

- ☒ Data
- ☐ Source Code
- ☐ Compiler/Linker
- ☐ Operating System
- ☐ Hardware
- ☐ Infrastructure

**Expertise Required to Implement:**

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☒ Custom Programmer (General Knowledge)
- ☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

- ☒ Seamless
- ☐ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** Handling encrypted webpages requires implementation in the webserver or a man-in-the-middle web proxy.

**Weaknesses:** This technique assumes that web bots remain relatively unsophisticated, and rely solely upon HTML metadata in form submission. While this is largely true in today's web bots, this lack of sophistication is only because it is currently unneeded, not because it is particularly complex to implement. Analysis of labels, locations, the Document Object Model, expected behaviors, and cross-request comparisons are all relatively straightforward paths to circumventing this defense. Because the intent is to provide a seamless experience for human users, there are sharp limits on the randomization that may take place. In particular, any given site can be automated with only a moment or two for a human operator to identify key features and turn the bot loose. The defense is best implemented as a proxy, but cannot randomize encrypted web pages — a common feature in web forms — from that location.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** Attackers cannot use the simple web bots that are currently widely used, and must upgrade to more complex web bots.

**Availability:** Not publicly available

**Additional Considerations:** None

**Proposed Research:** None

**Funding:** Qatar National Research Fund

### 2.7 HERMES

**Defense Category:** Dynamic Data

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Cross-Virtual Machine Cryptographic Key Theft

**Details:** This technique [13] is intended to prevent attackers from stealing public keys used to create SSL/TLS (and other) cryptographic keys from uncompromised virtual machines in environments where multiple virtual machines, some potentially malicious, are hosted on the same shared hardware infrastructure.

**Description:**

**Details:** This technique splits cryptographic keys across multiple virtual machines (VMs) using random shares that periodically rotate. The purpose of this action is to prevent cross-VM attacks that can infer the memory contents of a VM resident on the same hardware. Such attacks have been demonstrated as being practical. The random shares are created with Distributed RSA and Threshold RSA, ensuring that the individual shares are valueless on their own, and re-create and re-distribute the shares periodically to thwart long-term attacks. It is implemented as a library function that is transparent to the main application for SSL connections. This technique is intended to run on large distributed cloud systems (such as EC2) where multiple VMs are expected and the environment is relatively unconstrained. Individual session keys are not protected because of their short-duration lifespan combined with the difficulty of maintaining a persistent session without immediate access to the key at all times. Keys are rotated on a wall clock cycle, customizable by the operator and tested with periods from 5 to 125 seconds by the developers.

**Entities Protected:** Public cryptographic keys

**Deployment:** An updated version of then OpenSSL library, in this implementation

**Execution Overhead:**

- Key resharing takes up to 50 milliseconds.

**Memory Overhead:**

- Not discussed, but likely negligible

**Network Overhead:**

- Proportional to number of connections up to network saturation

**Hardware Cost:**

- None, but performance improves with additional VM resources

**Modification Costs:**

☒ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☒ Persistence

**Interdependencies:** None beyond cloud deployment requirements

**Weaknesses:** This technique has two primary weaknesses. First, there has been no analysis of how much time is necessary to steal cryptographic key shares from a VM. The key-rotation interval of 5 to 125 seconds is explained purely in terms of performance rather than security, and it is unclear as to what amount of wall clock time is actually necessary, or even indeed whether wall clock time is the correct measure in the first place. Without this analysis, the efficacy of this approach against real-world exploits is undetermined. Second, the initial system bootstrapping process requires uncompromised SSL channels between the virtual machines in order to establish the initial key share. If those channels are compromised, or if the share can be delayed until the key can be stolen from the initial VM, compromise is achieved. A minor weakness is that VMs continue to use old shares until all VMs have confirmed receipt of the new shares upon a rekeying cycle. If an attacker can delay the acknowledgment of a new key by one or more VMs, the old shares continue in use and remain present in memory.

**Types of Weaknesses:**

- ☐ Overcome Movement
- ☐ Predict Movement
- ☒ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** Attackers who encounter this system face a much more difficult task to acquire cryptographic keys.

**Availability:** No code is publicly available.

**Additional Considerations:** None

**Proposed Research:** Improvements to the bootstrapping process and a formal analysis of appropriate key rotation times would significantly advance cross-VM security methods not only for cryptographic keys, but for other sensitive data.

**Funding:** Air Force Office of Scientific Research, National Institutes of Health Grants, National Science Foundation, Army Research Office

## 2.8 CONTENT RANDOMIZATION OF MICROSOFT OFFICE DOCUMENTS

**Defense Category:** Dynamic Data

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Malicious Microsoft Office Documents

**Details:** This technique [14] is designed to block certain attacks embedded in Microsoft Office OLE- and OOXML-formatted documents. It can be employed at any time in the lifespan of the document.

**Description:**

**Details:** This technique randomizes the layout of OLE- and OOXML-formatted Microsoft Office Documents to break certain embedded exploits while preserving document structure and logical connections. It takes advantage of the fact that OLE and OOXML formats are containerized and their components can be freely reordered without breaking the ability of Office to parse and manipulate the document. Because malicious embeds inside Office documents frequently depend upon a particular ordering of internal components, the randomization prevents the exploits from executing. Content randomization can occur at any point, including when saving the document, when opening the document, or at any intermediate point between those events. The technique is not perfect and does not block all malicious documents, but is relatively non-invasive and has only minor compatibility interference. Randomization of the (legacy) OLE format is more comprehensive and effective than that of the (newer) OOXML format, but both offer protections.

**Entities Protected:** Microsoft Office applications

**Deployment:** This technique can be implemented in locations as convenient, including end-point machines opening the documents or on web gateways or file servers hosting such documents.

**Execution Overhead:**

- Randomization time of a document is minor; it is approximately comparable to an antivirus scan of the file.
- No additional time is required to open an OLE document.
- About 2.9% more time is required to open an OOXML document.

**Memory Overhead:**

- Not studied, but likely negligible

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☒ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** None

**Weaknesses:** This technique does not block all malicious documents, but only certain types of them. Additionally, it may be possible to modify some of the blocked documents to reduce the complexity of the exploit and shrink the footprint, such that randomization does not occur at the level required to break an attack. It may be possible to modify the technique's container size to block smaller fragmented malicious payloads, but no information is available on the countermeasures that malware could take in response. The authors also note that certain documents appear to fail the Office integrity checks and thus open in Protected View, which does not compromise their content but indicates that some trait is being missed in the randomization.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** Attackers would no longer be able to use most common types of malicious Microsoft Office documents and would need to modify the payloads or seek other avenues of attack.

**Availability:** No implementation is publicly available.

**Additional Considerations:** None

**Proposed Research:** This technique could benefit from additional research to improve its robustness and study the threat model, but the basic idea is sound and the claims are not overstated.

**Funding:** Lockheed Martin Corporation, National Science Foundation

# 3. DYNAMIC SOFTWARE

## 3.1 CCFIR: COMPACT CONTROL FLOW INTEGRITY AND RANDOMIZATION

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** This technique [15] prevents an attacker from redirecting code-flow to arbitrary locations, which is a necessary attacker capability for control-hijacking attacks.

**Description:**

**Details:** Compact Control Flow Integrity and Randomization (CCFIR) is a method of protecting commodity Windows executables. It is based on a set of modules that identify potential control-transfer targets, perform binary rewriting and randomization, and security policy validation. It is a user-space technology and does not rely on source code being available in order to provide protection. Programs are loaded and disassembled in order to identify all indirect and direct jump/call/return targets. Valid targets are redirected to code stubs in a special area of memory that is isolated from the regular program code sections. These stubs are aligned in certain ways to tag them (as function pointer stubs, sensitive return address stubs, and normal user return stubs.)

**Entities Protected:** This technology protects commodity user-mode Windows executables without requiring access to source code.

**Deployment:** CCFIR needs to be installed on end-user system and run against any executable to be protected.

**Execution Overhead:**

- Benchmarked execution overhead averaging 3.6% with a maximum of 8.6%

**Memory Overhead:**

- Some additional but minimal physical memory is required for the CCFIR Springboard code sections storing direct target jump stubs.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification required.)

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☒ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique depends on ASLR.

**Weaknesses:** If CCFIR is incrementally deployed (not protecting the entire system), pointers to unprotected modules can still be modified by an attacker, allowing a redirection of control flow and traditional ROP-style attacks. Moreover, CCFIR is also vulnerable to control flow hijacking because of its coarse granularity [16].

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** If fully deployed, attackers cannot perform Turing-complete return-to-library attacks. Additionally, unintentional instruction gadgets can no longer be used for ROP, making attacks significantly more difficult.

**Availability:** No code is publicly available.

**Additional Considerations:** None

**Proposed Research:** Weaknesses arise in control-flow integrity owing to the imprecision of the analysis that determines which control-flow transfers are valid. This technique is implemented as a binary-translation tool, which means the control-flow graph must be generated from the binary. If similar functionality were implemented in the compiler, a more precise control-flow graph could be used.

**Funding:** AFOSR, DARPA, National Natural Science Foundation of China, NDRC InfoSec Foundation, ONR

### 3.2 SOFTWARE DIVERSITY USING DISTRIBUTED COLORING

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection

**Details:** This technique [17] reduces the number of machines an attacker can successfully compromise in a network using code injection attacks. It does not prevent individual machines from being compromised.

**Description:**

**Details:** This meta-technique involves taking existing code diversity techniques and applying them across an entire network. The authors attempt to answer the following question: assuming that an adversary must specially craft an attack for each version of a diverse executable, and we have access to $k$ versions of an executable, how can we place these versions on a network so as to minimize the number of compromised machines (conversely, maximize the effort of the attacker)? Since we are trying to minimize the number of connected machines running the same version, this is the same as asking for an optimal $k$-coloring of the graph representing our network. Unfortunately, finding the minimum number of colors needed for a perfect coloring of a graph (such that no connected nodes are the same color) is NP-hard, as is the problem of finding an optimal coloring using $k$ colors. Instead, they propose a distributed heuristic approximation algorithm that results in at most $n/k$ links between two nodes of the same color, where $n$ is the number of nodes. If $k \geq n$ then each node can have its own color (version of the software) and the attacker will require a new custom attack for each node. If it is lower, then an attacker will only be able to infect a new node at each step with probability approximately equal to $1/k$. This gives us good utility out of the diverse executables that we do have.

**Entities Protected:** The overall network is protected from easy compromise by an attacker.

**Deployment:** The approximation algorithm used for assigning versions is distributed meaning that it must be run on every computer in the network. It could also be deployed from a centralized server that is distributing software to the network.

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification is required)

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique relies on already having diversified versions of the applications available. Other diversification techniques must be available for this technique to be useful.

**Weaknesses:** The proposed idea is more a planning tool than a stand-alone technique. Also even assuming that diversity can stop large-scale attacks, this method does not stop attacks against one machine.

**Types of Weaknesses:**

☐ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☒ Disable Movement

**Impact on Attackers:** If the underlying diversity used is sound, then this technique makes it harder for an attacker to compromise an entire network using only one attack. Depending on the number of software versions available, he could be limited to a small portion of the network.

**Availability:** None, results only theoretical

**Additional Considerations:** This is more a planning method that a stand-alone technique. The results are highly theoretical.

**Proposed Research:** The actual impact of diversity on successful attacks must be studied and analyzed.

**Funding:** National Science Foundation, Koerner Family Fellowship

### 3.3 SECURITY AGILITY FOR DYNAMIC EXECUTION ENVIRONMENTS

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Exploitation of Trust

**Details:** This technique [18] aims to mitigate system and network intrusions at a high level by dynamically modifying security policies.

**Description:**

**Details:** The authors describe and implement a software toolkit that allows applications to be developed around the idea of dynamically changing security policies. The main problem with moving from static security policies to dynamic policies is that unmodified applications will not be able to adjust to policy changes that leave them without access to crucial resources. The authors introduce a framework for designing applications with multiple behaviors that can transition from one to another depending on which resources (both on the same machine and on the network) are available under the current security policy. This allows security policies to change on the fly, in response to an actual or attempted intrusion, while maximizing the utility of the machines and applications on the network at all times. An agile policy controller that can set and modify the security policies over the whole network dictates the security policies on each machine.

**Entities Protected:** Protects the network from potential intrusions and provides a way of mitigating successful intrusions.

**Deployment:** Requires deployment on all machines in a network as well as at least one additional policy controller.

**Execution Overhead:**

- Varies depending on the application: backup behaviors could be less efficient in order to get around reduced resources of some security policies

**Memory Overhead:**

- Varies depending on the application: backup behaviors could be less efficient in order to get around reduced resources of some security policies

**Network Overhead:**

- Varies depending on the application: backup behaviors could be less efficient in order to get around reduced resources of some security policies

**Hardware Cost:**

- Requires at least one additional policy controller machine

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique relies crucially on a detection capability. This can be very challenging for polymorphic type attacks [19,20]. If the attacks are not detected, they cannot adjust the policy.

**Weaknesses:** The policy manager becomes a new point of weakness since it can dynamically change the security policies of all the other machines on a network. The authors provide a mechanism for distributing the duty amongst several machines so that no single point of trust exists.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** This technique makes it more difficult for an attacker to advance an intrusion due to the network security policies reacting dynamically to his or her attack.

**Availability:** Research was done as part of a DARPA project so we assume the code is available.

**Additional Considerations:** This work lacks many specifics. For example, how the policy is adjusted or what impact policy adjustment has on the system. See [71] for more on dynamic policy.

**Proposed Research:** The actual impact of agility and policy adjustment on the security posture of a system must be studied. Also reliance on a perfect detection capability must be relaxed in such a system.

**Funding:** DARPA

### 3.4 PROACTIVE OBFUSCATION

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [21] aims to mitigate buffer overflows and other injection attacks on network visible services.

**Description:**

**Details:** The authors use a similar technique to DieHard but in a more generalized setting. Since control injection attacks have to be individually tailored to specific executables, this technique creates multiple copies of each service executable, randomized differently. The randomization used can be any of the other executable randomization techniques we have described such as ISR, ALSR or system call randomization. Whenever a request is issued to the service, it is multiplexed to each of the replicas and the responses are tallied like a vote. If a majority of the replicas agree, then the response is sent out. The idea is that any attack should only work on one of the replicas and the others will remain uncorrupted, so a majority vote will result in a correct response. However, it is more likely that one will be compromised and the others will crash (due to different addresses, system calls, *etc.*). This means that if the system returns a response, it will be correct with a high degree of certainty but it may not answer if a majority of the replicas have crashed. In order to prevent an attacker from gaining some progressive knowledge and eventually letting him compromise all the replicas at once, the system proactively reboots replicas with new randomization. There is a controller that dispatches the requests and tallies votes, as well as controls when replicas will be rebooted (a configurable time limit).

**Entities Protected:** Servers

**Deployment:** Can be deployed on any server with important trusted services.

**Execution Overhead:**

- Experimental execution overhead of 20% (differs depending on application, this estimate is very optimistic) and latency overhead of 40%

**Memory Overhead:**

- Extra memory must be used to store the multiple running replicas so an M times memory overhead where M is the number of replicas

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This method does not propose a new randomization technique and relies on existing diversification techniques.

**Weaknesses:** The controller that dispatches and maintains replicas is now a new target for attack, since it is a single point of failure. Additionally, a single compromised replica can destroy it if it is not also replicated. Also this technique does not protect against information leakage (exfiltration) that happens on one replica.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☒ Disable Movement

**Impact on Attackers:** The technique on its own does not provide protection. It relies on existing randomization techniques and voting.

**Availability:** No publicly available code.

**Additional Considerations:** The technique on its own does not provide protection. It relies on existing randomization techniques and voting.

**Proposed Research:** This technique ensures correct responses by voting amongst the replicas, but it does not ensure that individual replicas cannot cause damage locally. If multiple replicas were running on the same machine, the operating system interface (system calls) could be considered as the other side of a container holding these replicas. Every time a single replica executes a system call, if the other replicas are uncompromised, they will also issue the same system call. If one of the replicas is compromised, it must deviate from proper behavior by calling a different series of system calls that can be detected as aberrant. If it does not deviate, then it cannot do anything useful. Therefore, the operating system could only execute system calls if a majority of the replicas request the same system call, ignoring all others.

**Funding:** Air Force Office of Scientific Research, National Science Foundation, Microsoft Corporation

### 3.5 PROGRAM DIFFERENTIATION

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection and Code Injection

**Details:** This technique [22] mitigates buffer-overflow attacks on remote services.

**Description:**

**Details:** The authors aim to design a secure mobile-phone platform that is not vulnerable to remote attack through buffer overflow exploits. They note that buffer overflow attacks can be defended against using several different orthogonal techniques to increase effectiveness. One of these techniques, system-call randomization, is old and two more are unique to this paper.

The authors propose that, since mobile platforms are rapidly evolving, it may be useful to consider hardware changes that could defend against buffer overflow attack. Toward this, the first defense they propose is modifying the return instruction. The vulnerability in the return instruction is that it returns to an address specified on the stack, which can be targeted by an attacker. Instead, the new return address will only take an index into a table that contains the actual return addresses. This table will be readable only by the return instruction and writable only by the call instruction, so it will not be vulnerable to inspection or tampering. At the start of a function call, the call instruction will insert the return address into this table with a random unused index. It then puts this index on the stack. The return instruction loads the actual address from the table based on the index on the stack and jumps to the specified location. The address table is protected so that it can only be read by the return instruction and written to by the call instruction.

The second technique the authors propose is to use instruction packing to differentiate at the instruction set level. The way instruction packing works is it compresses frequently used instructions together into one instruction with an Instruction Register File (IRF). This IRF stores the instructions in an indexed table and when the program wishes to use a sequence of these instructions, it can instead call a 5-argument pack instruction with the indices of the instructions it wishes to use. For instance, if an often-used sequence of instructions is stored in the table with indices 1-5, the program would invoke all five instructions at once with a single instruction *pack5* 1 2 3 4 5. If the indices of the IRF are randomized, then this creates a unique instruction set for each executable.

**Entities Protected:** This scheme is targeted at mobile platforms but could be used anywhere the custom hardware was available.

**Deployment:** Deployed at the local machine level by modifying hardware.

**Execution Overhead:**

- Unknown execution overhead due to additional table lookups

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- Requires special hardware with the modified instruction set described

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☒ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** This method should be combined with a ROP defense.

**Weaknesses:** The method is vulnerable to return-oriented programming without returns since the jump instruction is not protected.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** Makes it very difficult for an attacker to inject code (since he or she cannot guess the correct indices into the IRF) and impossible for an attacker to return to arbitrary locations in the code. ROP is still possible though.

**Availability:** The hardware specified does not actually exist yet.

**Additional Considerations:** The technique is effective against traditional code injection, but the hardware modification proposed makes it impractical for existing systems.

**Proposed Research:** A complete code injection and ROP protection method is an open problem.

**Funding:** National Science Foundation

### 3.6 PROGRAM PARTITIONING AND CIRCUIT VARIATION

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [23] aims to prevent exploitation of traditional software vulnerabilities by moving vulnerable code into hardware circuitry.

**Description:**

**Details:** The authors describe a system in which a traditional application is partitioned into two components. A general purpose processor component runs non-vulnerable sections of the code on traditional computer hardware. Sections of the code that may be vulnerable to traditional vulnerabilities such as buffer overflows and other address/program counter related exploits can be translated into logic blocks using hardware description languages and are targeted to run on an FPGA. A state-transfer interface is used to pass data from the general purpose processor to the FPGA and vice-versa. Further security can be obtained through dynamic variation of the circuits running on the FPGA.

**Entities Protected:** This technology protects applications written in C or other high-level languages (HLL) that can be translated into hardware description languages (HDL).

**Deployment:** To deploy this approach, a program must be partitioned and portions of the source code must be ported either manually or with the help of an HDL-to-HLL translating compiler, targeted to a specific FPGA paired with a general-purpose/commodity-computer platform.

**Execution Overhead:**

- Unknown execution overhead due to data-passing interface between general-purpose processor and FPGA, potentially lower performance of algorithms implemented on FPGA compared to general-purpose hardware.

**Memory Overhead:**

- Negligible overhead for storage of data to be passed between general-purpose processor and FPGA.

**Network Overhead:**

- None

**Hardware Cost:**

- Additional hardware needed (Xilinx Virtex-5 FPGA referenced in paper ranges from $100–$1000.)

**Modification Costs:**

☒ Data

☒ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☒ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique can benefit from the addition of ASLR and ROP protection techniques to protect the sections of code that are resident on the general-purpose processor partition.

**Weaknesses:** Programs may be improperly or insufficiently partitioned, leaving vulnerable sections of code running on the general-purpose processor and exploitable by traditional attack techniques. Attacker may be able to insert vulnerabilities into the FPGA circuit, allowing non-traditional routes of exploitation.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☒ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** Attackers cannot target addressing-related vulnerabilities in software components that are running on hardware without the concept of an address space or a program counter, preventing many traditional avenues of attack. Attackers would need to focus on exploiting the dynamic hardware itself.

**Availability:** It is not clear that the technique has actually been implemented.

**Additional Considerations:** FPGAs are bootstrapped at startup or system reset. This limits the ability to swap in a new program circuit variant without taking down the entire system. Additional hardware may present additional attack vectors.

**Proposed Research:** Investigate the increased vulnerability space presented through new hardware additions such as the integration of Altera FPGA into the commodity Intel Atom product line.

**Funding:** National Science Foundation

### 3.7 LIBRANDO

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** This technique [24] protects against certain JIT-specific attack vectors including predictable code generation and JIT spraying, where an attacker injects short sequences of code into executable memory through constants in High Level Language source code that is JIT compiled and later utilized in conjunction with other exploits to cause arbitrary code execution.

**Description:**

**Details:** This work aims to deter successful attacks against JIT compilers through multiple dynamic code-diversification techniques. NOP insertion is used to randomize the code that a JIT compiler emits for a given high-level-language construction. Constant blinding is used to prevent attackers from being able to emit the equivalent of valid, arbitrary attacker-chosen machine code instructions into executable memory by having constants present in the source code encrypted before they are emitted into the output native executable code and decrypted through a few extra emitted native code instructions. These methods are implemented as a standalone library, `librando`, which can be used as a black box with unmodified JIT compilers or as utility library within the compiler source itself.

**Entities Protected:** Any application that performs JIT compilation/code-generation with or without source code available.

**Deployment:** Applications that perform JIT compilation can run unmodified and load `librando` through `LD_LIBRARY_PRELOAD` or they can have their source code modified, recompiled, and linked with `librando`.

**Execution Overhead:**

- Ranging from 1.1x for best-case applications run by a JIT compiler using `librando` as a utility library up through 13x for worst-case applications run by JIT compiler using `librando` as a blackbox.

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

- ☐ Data
- ☒ Source Code
- ☒ Compiler/Linker
- ☐ Operating System
- ☐ Hardware
- ☐ Infrastructure

**Expertise Required to Implement:**

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☐ Custom Programmer (General Knowledge)
- ☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

- ☐ Seamless
- ☐ Simple Configuration
- ☒ Complex Configuration (System Admin)
- ☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** `librando` is best used in conjunction with other diversification solutions for the static code of an application along with techniques such as ASLR or code randomization and ROP protection.

**Weaknesses:** `librando` only diversifies code that cannot be protected through the use of ahead-of-time solutions. Use of `librandp` alone leaves the static program code unprotected and vulnerable to exploitation by an attacker.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** Attackers will not be able to use JIT-spraying and predictable code generation to force a JIT compiler to output useful gadgets as part of an exploit attempt, forcing them to rely much more heavily on vulnerabilities in the static code base of the application.

**Availability:** No code publicly available.

**Additional Considerations:** Protections on dynamically generated code are useful but without protecting of the static code comprising the JIT compiler and the rest of the application containing the JIT compiler, there are many more possible opportunities for application exploitation.

**Proposed Research:** Additional randomization techniques could be added to `librando` in order to enhance security in certain situation. Memory-map randomization in the absence of ASLR, increased code-randomization granularity, instruction replacement and instruction recording could be used to provide more sources of randomization in dynamically generated code.

**Funding:** Defense Advanced Research Projects Agency, National Science Foundation, Google

### 3.8 REDHERRING

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Data Leakage, Resource, Injection, Privilege/Trust

**Details:** This technique [25] aims to disrupt attackers by redirecting exploit attempts to unpatched software decoys that give the appearance of a successful attack and allow monitoring, while offering the security of a conventional patch.

**Description:**

**Details:** The authors present a system whereby a lightweight virtualization mechanism is used along with a small library for implementing patches to present attackers with a sanitized decoy environment when they attempt to exploit a patched vulnerability. The decoy environment is forked from the main application and presents the attacker with the appearance that their attack has succeeded. This allows the monitoring of the full attack payload while preventing access to actual sensitive data. Meanwhile, the patch prevents exploitation in the main application and execution continues as normal for other users.

**Entities Protected:** Any application with source code available.

**Deployment:** Application source code/patch modifications are made using the honey-patch library and the rebuilt application is deployed to the server along with a small set of services (virtualization controller, checkpoint controller, checkpoint server, reverse proxy).

**Execution Overhead:**

- Maximum observed 2.6x overhead due to session forking and memory redaction when an attack triggers a honey-patch.

**Memory Overhead:**

- None

**Network Overhead:**

- Minimal additional round-trip latency added to network service requests due to execution overhead.

**Hardware Cost:**

- None

**Modification Costs:**

- ☐ Data
- ☒ Source Code
- ☐ Compiler/Linker
- ☐ Operating System
- ☐ Hardware
- ☒ Infrastructure

**Expertise Required to Implement:**

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☒ Custom Programmer (General Knowledge)
- ☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

- ☐ Seamless
- ☐ Simple Configuration
- ☒ Complex Configuration (System Admin)
- ☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☐ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** Relies on prior knowledge of vulnerabilities in software to be protected so that patches may be developed.

**Weaknesses:** Provides no protection against zero-day/unknown software vulnerabilities in an application. Attackers with control of decoys can potentially reverse-engineer process image and discover patches.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☒ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** Makes it difficult for attackers to determine if their attack attempts have succeeded. Disinformation provided by decoy environments may frustrate further attempts to move beyond application/machine being attacked.

**Availability:** No code publicly available.

**Additional Considerations:** Four additional services have to be run on the system alongside the patched application, providing a potentially increased attack surface. Further work can be done to improve security during decoy creation, such as unloading of the honey-patching library.

**Proposed Research:** None

**Funding:** ONR, AFOSR, NSF, CASED, EC-SPRIDE Darmstadt, BMBF

### 3.9 REVERSE STACK EXECUTION IN A MULTIVARIANT EXECUTION ENVIRONMENT

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection

**Details:** This technique [26–30] detects buffer overflows on the stack and prevents exploitation of them through stack smashing.

**Description:**

**Details:** The authors propose a very simple form of multivariant execution with two replicas where one replica runs with the stack growing upward and the other runs with the stack growing down. Normally, any single architecture only supports the stack growing in one direction, but the authors introduce a compiler transformation that can create a program with an opposite direction stack. Any buffer overflow attack that works on one would necessarily not work on the other because the overflow would be writing over different parts of the stack. Therefore, a divergence in behavior would signify that such an attack has occurred and the operating system could detect that and terminate the program.

**Entities Protected:** Any generic machine with this technique deployed in the compiler.

**Deployment:** Deployed on any machine by modifying the compiler and operating system.

**Execution Overhead:**

- 100% execution overhead to run a replica
- Experimental results show only a 3% overhead in the replica

**Memory Overhead:**

- Up to 20% increased executable size
- 100% memory overhead for an additional replica

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

- ☐ Data
- ☐ Source Code
- ☒ Compiler/Linker
- ☐ Operating System
- ☐ Hardware
- ☐ Infrastructure

**Expertise Required to Implement:**

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☐ Custom Programmer (General Knowledge)
- ☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

- ☒ Seamless
- ☐ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** This method should be combined with ASLR and ROP protection techniques for better results.

**Weaknesses:** A monitor is required to dispatch inputs to both replicas and to detect when their execution diverges. This monitor is itself vulnerable to attack as it has the same weaknesses as any other program. Additionally, there are some special cases where a buffer overflow can work on a stack in both directions equally. Specifically, if a buffer overflow occurs and there is no system call between it and the return function.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** Attackers must find a weakness in the monitor or a more specific type of buffer overflow.

**Availability:** No code publicly available.

**Additional Considerations:** It requires source code of any application to be protected. It also requires an additional replica to be run (100% execution overhead). Similar, but more limited multi-variant techniques have been proposed [31].

**Proposed Research:** An improved technique can use a similar method but without relying on replicated execution.

**Funding:** Unknown

### 3.10 GENPROG: A GENERIC METHOD FOR AUTOMATIC SOFTWARE REPAIR

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** An attacker attempting to exploit known software vulnerabilities prior to the application of a patch. This technique [32] automatically generates patches for known bugs in C programs, limiting the time window for exploitation of such vulnerabilities.

**Description:**

**Details:** This work aims to decrease the patch delay by automatically creating patches for C programs with known bugs. Given source code, along with test cases that demonstrate both the bug as well as all required behavior of the program, GenProg creates a patched variant. Genetic programming techniques are used to find a patch that both maintains required functionality and eliminates the vulnerability. Statements are extracted from existing pieces of the source code and used for generating potential patched variants of the program. The variants are then tested against a fitness function that checks that the bug test case no longer succeeds and that the required test cases do succeed.

**Entities Protected:** Applications with C source code available.

**Deployment:** GenProg software needs to be installed on systems. Also requires source code and test cases for the target applications.

**Execution Overhead:**

- Negligible, often just a few instructions to implement a check for the exceptional case

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☒ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This method should be combined with ASLR and ROP protection techniques that protect against unknown vulnerabilities. It should also be considered within the context of larger systems that are able to detect vulnerabilities and trigger this technique to generate patches.

**Weaknesses:** The generated patch is not guaranteed to block all possible exploits for a given bug. The generated variant may pass the test case and block the exact exploit payload that it was provided but a small modification to the exploit could still result in arbitrary code execution. Additionally, the technique provides no protection against unknown vulnerabilities.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** The attack window is decreased as applications gain the ability to repair themselves without relying upon manual, time-consuming patch procedures.

**Availability:** Source code is available.

**Additional Considerations:** The test cases demonstrating the required functionality of the application must be accurate or patches may be generated that break the functionality of the program.

**Proposed Research:** Investigating mechanisms for automatically generating test cases for the required functionality. Applying this technique to software that lacks source code.

**Funding:** DARPA Clean-slate Resilient Adaptive Secure Hosts (CRASH) Program

### 3.11 DISTRIBUTED APPLICATION TAMPER DETECTION VIA CONTINUOUS SOFTWARE UPDATES

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Spoofing, Control Injection, Code Injection

**Details:** This technique [33] is designed to mitigate Remote Man-at-the-End (R-MATE) attacks. These are essentially attacks by malicious end-points that participate in a larger distributed system, such as many clients communicating with a shared server. The attacker, who has full access to the hardware and software on the end-point, seeks to tamper with the client binary currently executing without alerting the server to this fact. This technique relies on periodic code mutation to detect such tampering.

**Description:**

**Details:** This technique uses a remote variant of dynamic software diversity to detect code tampering. On the client-side, only a stub application is initially launched. As the client requires execution of new functions, the server will compile those functions using diversity transformations (*e.g.*, obfuscation, randomization) and send them to the client. The server uses control-flow analysis of the client binary to validate the block being requested is actually a valid control transfer from the current program state. The server can also push compiled functions to the client in anticipation of their use. Periodically, the server will regenerate all of the client's current code fragments in order to invalidate any blocks that have been tampered with.

**Entities Protected:** Benign end-points and servers in a distributed system

**Deployment:** This technique requires diversification agents to be installed on a central server. It also requires all client code to be re-written in order to support remote retrieval of code blocks.

**Execution Overhead:**

- Overhead from diversification was measured to be an average of 10%, and a worst-case of 20%, on the `bzip`, `gzip`, `mcf`, and `crafty` binaries. This does not include stalled execution waiting on blocks to be sent from the server. The authors estimate the additional overhead from compiling and sending code blocks is approximately 1.5 seconds per function, not counting network transport latency.

**Memory Overhead:**

- The authors do not provide any evaluation.

**Network Overhead:**

- The authors do not provide any evaluation. This could be substantial for large numbers of clients, however.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☒ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique depends on a remote server to provide diversified code, and requires a constant connection in order for the protected binary to execute correctly.

**Weaknesses:** This technique does not provide any analysis of how much difficulty its diversification scheme actually poses for an attacker. Depending on the mutation rate, attacks may succeed for a lengthy period of time before being detected, if at all. In addition, it relies on computing a control-flow graph to detect attacks relying on block requests. These have been shown insufficient to stop many code reuse attacks [34].

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This defense may require an attacker to periodically re-analyze the program in order to stressfully tamper with it. It is not clear this provides the attacker with a substantial disadvantage, however.

**Availability:** No code publicly available.

**Additional Considerations:** None

**Proposed Research:** This defense would benefit from a deeper security evaluation, especially with respect to what kinds of tampering can and cannot be detected, and how much cost/difficulty is imposed on the adversary. In addition, research to remove the always-connected requirement would make this defense much more practical for real-world applications.

**Funding:** Unknown

### 3.12 THWARTING CACHE SIDE-CHANNEL ATTACKS THROUGH DYNAMIC SOFTWARE DIVERSITY

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Data Leakage Attacks

**Details:** This technique [35], can mitigate side-channel-based data leakage. While the framework is generalizable to other side channels (*e.g.*, power), the authors focus on cache-based side channels between collocated VMs in a cloud environment. By adding dynamically changing levels of cache noise, the technique disrupts timing side channels based on measuring cache hits and misses. These attacks leak, to the attacker, what memory addresses are accessed by the victim. This information can be used to, e.g., derive cryptographic keys.

**Description:**

**Details:** This technique uses Dynamic Software Diversity, which adds randomness to program control flow in order to make it more difficult for attackers to discern what instructions are being executed, while retaining the semantics of the original code. It operates by creating, at compile-time, replicas of code fragments (which may be chosen either probabilistically or targeted at sensitive functions like encryption/decryption). Each of these replicas is put through a series of diversifying transformations, which add instructions (whose quantity and location vary per replica) to create noise on side channels. For example, random memory load instructions are used to disrupt cache side channels. At run time, whenever the program would execute a diversified code region, the program randomly chooses one replica to execute.

In order to ensure that attackers cannot simply profile the application to remove the noise, each of these memory load instructions is itself randomized. An asynchronous thread maintains a constantly changing table of memory addresses. Each randomized load instruction loads the memory address in one entry in this table. Thus, every time the cache noise is added, it is added to a different, randomly chosen cache line.

**Entities Protected:** This technology protects user-mode applications

**Deployment:** This technique requires protected applications to be recompiled

**Execution Overhead:**

- When 25% of the code base is dynamically diversified, there is an average 86% and worst-case 700% overhead on the SPEC 2006 CPU benchmark

**Memory Overhead:**

- The authors do not provide an analysis. At minimum, the overhead will be a function of both the number of replicas made and the number of functions diversified.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** None

**Weaknesses:** The authors tested their defense against PRIME+PROBE and EVICT+TIME side channel attacks, and found that it could reduce the number of leaked bits of a 128-bit key from approximately 100 to 16. However, this comes at a potentially substantial performance penalty, as discussed above. In addition, it may be possible to develop smarter cache side channel attacks that are aware of this defense. The noise added may also be removed, for example, as there is no cryptographic guarantee on the level of randomness added to the side channel.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique could hinder attackers trying to exploit cache-based side channels to leak sensitive data. The attacker would have to rely on other, more indirect, side channels such as branch prediction caches or disk access times.

**Availability:** No code publicly available.

**Additional Considerations:** None

**Proposed Research:** The technique is useful, but imposes potentially severe execution overhead. Research on optimized diversification to reduce this penalty, while maintaining security, would be valuable. Additionally, extending this approach to other side channels (*e.g.*, power) would increase its coverage.

**Funding:** Defense Advanced Research Projects Agency

### 3.13 MVARMOR

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection, Control Injection, Data Leakage

**Details:** MvArmor [36] mitigates control- and code-injection attacks. The threat model considers an attacker that can interact repeatedly with the target program, and is able to read and write to arbitrary addresses in memory. They have access to both relative and absolute memory operation primitives, and can use both spatial and temporal vulnerabilities. The authors consider both attackers attempting to gain arbitrary code execution, as well as attackers trying to leak sensitive data in memory.

**Description:**

**Details:** MvArmor relies on multi-variant execution, in which multiple diversified variants of the same program are run simultaneously. Any input to one is copied to all other variants. Their state is compared and synchronized on sensitive system calls (*e.g.*, I/O and memory manipulation calls). Between these events they execute independently. At synchronization time, a security monitor compares the arguments, system call, and return value across all variants in order to detect attacks.

In order to protect against memory errors, MvArmor generates variants with three properties, all related to memory layout. To deterministically stop absolute spatial attacks, non-overlapping address spaces are used. Every variant has an address space unique to it, so that any memory address stored in a pointer cannot ever point to mapped memory in another variant. Thus, corrupting that pointer would cause errors in all other variants. To deterministically stop relative spatial attacks (*e.g.*, buffer over-reads) non-overlapping offset spaces are used. This ensures that the relative distances between any two objects is unique to a variant, and attempting to use, *e.g.*, a partial pointer overwrite will fail in all other variants. Finally, to protect against temporal attacks (*e.g.*, use-after-free exploits), MvArmor uses a combination of randomized allocators and approximate type-safe address reuse. This probabilistically stops temporal exploits. The authors note that deterministically preventing such exploits requires recompilation, which they do not assume is available.

**Entities Protected:** User-space applications converted using MvArmor

**Deployment:** MvArmor requires a binary to protect, but no source code. It also relies on hardware-assisted virtualization, such as Intel VT-x. The implemented version requires Dune [37], which is Linux-specific and used to implement process virtualization.

**Execution Overhead:**

- MvArmor was evaluated on the SPEC 2006 CINT benchmark. It has an average overhead of 9.1% for two variants, and 20.4% for four variants.

**Memory Overhead:**

- The authors do not evaluate memory overhead. At minimum, it will be the product of the code size and the number of variants

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** This technique depends on ASLR being present in order to implement variant generation.

**Weaknesses:** MvArmor focuses many of its protections on the heap. Attacks using other regions of memory (*e.g.*, the Global Offset Table) may be able to bypass protection from temporal and relative spatial attacks. In addition, attacks not relying on the set of sensitive system calls will not be detected.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** MvArmor imposes significant difficulty on attackers trying to leverage memory errors in order to leak data or achieve code execution.

**Availability:** No code publicly available.

**Additional Considerations:** None

**Proposed Research:** Further evaluation of attacks using regions of memory outside the stack and heap (*e.g.*, Global Offset Table) is needed to determine the security guarantees of this system.

**Funding:** European Commission, Netherlands Organization for Scientific Research

This page intentionally left blank.

# 4. DYNAMIC RUNTIME ENVIRONMENT

## 4.1 ADDRESS SPACE RANDOMIZATION

### 4.1.1 ADDRESS SPACE LAYOUT PERMUTATION

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [38] defends against buffer overflow attacks on the stack and heap from an adversary that can provide arbitrary input to a vulnerable program. A buffer-overflow attack occurs when an attacker can provide malformed input to a program that causes it to write the input incorrectly to areas outside the allotted memory location. This technique defends against direct overflow attacks, where the goal is to overwrite the return pointer on the stack, and indirect attacks where the goal is to overwrite a function pointer on the heap that is later dereferenced. It does not protect against adversaries that have local access to a machine.

**Description:**

**Details:** This technique performs stack randomization at both the user and kernel levels. User-level permutation includes both a coarse randomization (code and data segments are randomly placed) and a fine-grained randomization (functions and variables are randomized inside code and data segments). The user-level permutation is implemented as a binary rewriting tool that processes Executable and Linkable Format (ELF) executables and outputs a randomized version with the same behavior. This rewriting does not require source code access or recompilation. At the kernel level, the starting location of the user stack is randomly chosen and the heap is removed from its usual place inside the data section and randomly placed in program memory. Additionally, the `mmap()` function is patched so that individual pages inside the heap are randomly allocated.

**Entities Protected:** All programs running on the machine are protected from code or control injection through individual, independent program randomization.

**Deployment:** This technique could be deployed on any generic machine.

**Execution Overhead:**

- The required kernel changes do not affect performance to a significant degree and user-level changes occur as a preprocessing step and so do not affect execution speed.

**Memory Overhead:**

- Experimental results show an approximately 20% increase in executable size and memory footprint.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** Memory randomization is more effective when it is combined with various types of memory guards [39, 40].

**Weaknesses:** As with many other address randomization techniques, the entropy of this scheme is limited [41, 42] by the architecture machine width (*i.e.*, number of bits: 32 or 64). In this case, they do get very close to that limit with 29 bits of entropy for the heap location, 28 bits for the stack location, 20 bits in `mmap()` and 20 bits within the data and code segments. This far exceeds other related schemes. However, their scheme is not resistant to attacks that can violate "memory secrecy" [43] through leakage or local access. It cannot randomize inside of stack frames so it is also vulnerable to return-oriented programming (ROP) attacks. It may also be vulnerable to a heap spraying technique [44] where large chunks of memory are allocated quickly to try to reduce uncertainty on the heap.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☒ Limit Movement

☐ Disable Movement

**Impact on Attackers:** Attacker level of effort is raised substantially. Although it may be possible to mount attacks using address leakage, it would require additional effort that is much higher than finding and exploiting the original buffer overflow.

**Availability:** Source code from the original researchers is available, which runs on Linux with ELF executables. Additionally, derivatives implementations from this original work are available in every major modern operating system, including Windows, Linux, OS X, iOS, and Android, among others.

For legacy Windows applications that were not built with the relocation information necessary to support address space layout randomization, recent research has demonstrated how that information can be dynamically reconstructed thereby allowing the address space to be randomized [45].

**Additional Considerations:** When applying this technique, it is important to ensure that each process is randomized differently. In earlier version of Android, this technique was applied, but due to the process by which applications were forked from a single *zygote* process, the libraries, while randomly placed, were placed uniformly in the address space of every application [46]. This allowed an attacker to glean the location of code in one application, and exploit it in another. To remedy

this problem, a pool of zygote processes, called a *morula*, can be maintained, thereby allowing a process with a different memory layout to be spawned quickly [46].

**Proposed Research:** Developing a memory-protection technique that does not assume memory secrecy and provide high entropy is an important missing piece.

**Funding:** Unknown

### 4.1.2 DIEHARD

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code and Control Injection

**Details:** DieHard [47, 48] protects the heap from indirect buffer overflow attacks where an attacker attempts to overwrite a function pointer to cause control injection.

**Description:**

**Details:** DieHard attempts to defend against four classes of vulnerabilities that could lead to program crash or code/control injection: invalid frees, buffer overflows, dangling pointers and uninitialized reads.

The strategy used has three main elements: address randomization, heap spacing, and replication. Addresses of heap objects are randomized using a different seed each time the program is executed. Additionally, the heap is sized to be $M$ times larger than is necessary for program execution. This allows for extra space between objects so it is less likely that a buffer overflow will result in overwriting of another object. DieHard also maintains $N$ copies of the heap initialized with different random seeds. Whenever a memory operation is done, a "vote" occurs between the copies. These three techniques together provide a probabilistic measure of defense against the four classes of vulnerabilities. Since there are multiple copies with different randomized addresses, any targeted buffer overflow would end up segmenting the control flow (*i.e.*, replicas would end up executing different segments of code). This would be discovered and a recovery mechanism could possibly be used [49–51].

**Entities Protected:** Can be configured to protect any or all programs on a machine.

**Deployment:** This technique could be deployed on any generic machine by patching the operating system.

**Execution Overhead:**

- Experimental results show an execution overhead of 50-100% with $M = 2$ and 3 replicas.

**Memory Overhead:**

- Because of the increased heap size and replicas, the memory overhead is quite large, at least $M * N$.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☒ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** DieHard and ASLR can interfere with each other and potentially have negative impact. DieHard consumes a large amount of memory, which makes the ASLR less effective.

**Weaknesses:** Provides only probabilistic security; depending on the parameters chosen, a system might be vulnerable to a brute-force attack. It also assumes "memory secrecy."

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** Makes it difficult to mount injection attacks against the heap and even more difficult to ensure that a given attack will work 100% of the time.

**Availability:** The code is available online for non-commercial use. A demonstration that is configured to provide heap randomization to Mozilla Firefox in Windows is also available.

**Additional Considerations:** In the process of stopping buffer overflow attacks, this technique also allows programs to recover from many common errors without crashing (See [52] for failure-oblivious computing). Most other memory-randomization techniques will prevent an attacker from gaining control, but will still cause the program to crash upon attempted exploitation of a buffer overflow.

**Proposed Research:** A low-overhead memory protection technique that does not assume memory secrecy is still an open research problem. The memory overhead of DieHard is really significant.

**Funding:** National Science Foundation, Intel Corporation, Microsoft Research

### 4.1.3 INSTRUCTION-LEVEL MEMORY RANDOMIZATION

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [53] defends against buffer-overflow attacks on the stack and heap from an adversary that can provide arbitrary input to a vulnerable program. A buffer overflow attack occurs when an attacker can provide malformed input to a program that incorrectly causes it to write the input to areas outside the allotted memory location. This technique defends against direct overflow attacks, where the goal is to overwrite the return pointer on the stack, and indirect attacks where the goal is to overwrite a function pointer on the heap that is later dereferenced. It does not protect against adversaries that have local access to a machine.

**Description:**

**Details:** This technique randomizes both the stack and heap. The randomization takes the form of a program that transforms an executable into a randomized version that has the same behavior. Random padding is added at the start of the stack and before the return address in every stack frame by modifying the assembly code that creates these stack frames. The placement of heap chunks is also randomized by requesting a chunk much larger than is needed and then placing the original chunk randomly inside that larger chunk. The main advantage of this technique is that it does not need access to source code or recompilation of target programs. It matches with the current software distribution model in that it could be hooked into an installer application that would randomize the executable differently for every machine where it is deployed.

**Entities Protected:** Any or all programs running on a machine that have been processed by the binary rewriter.

**Deployment:** Can be deployed to any generic machine as part of a platform configuration or individual programs can be manually randomized. This method is a separate application and does not require modification to any other component.

**Execution Overhead:**

- None

**Memory Overhead:**

- Stack and heap size increased by approximately 20%.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification required)

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique can be complimentary to ASLR, but it can have a conflict with DieHard. Applying both DieHard and this technique can make the memory overhead very large.

**Weaknesses:** This scheme only partially protects against ROP. It makes it more difficult to put arguments onto the stack that will be passed to the target library function, but does not fully prevent redirection of program control. The randomness injected is also limited by the machine architecture, namely it cannot be more than 32 bits (and it probably much lower than that in practice). They also cannot rewrite some instructions, so in their experimental results, they only protected about 70% of each executable. This technique is not effective against attacks that violate memory secrecy and may be vulnerable to heap spraying.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** Increases the level of effort for attackers in many circumstances. Since some instruction sequences cannot be processed by this technique, portions of executables may remain vulnerable.

**Availability:** No code publicly available.

**Additional Considerations:** This approach is notably different from many other memory randomization techniques in that it is done as a binary rewriting. This means that it could actually be installed on a software distribution server that would uniquely randomize executables as they were being distributed (and thus require no configuration or changes of any kind on the client).

**Proposed Research:** A low-overhead memory-protection technique that does not assume memory secrecy is still an open research problem.

**Funding:** National Science Foundation

### 4.1.4 OPERATING SYSTEM FINE-GRAINED ADDRESS SPACE RANDOMIZATION

**Defense Category:** Dynamic Runtime Environments

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection, Control Injection, and Data Leakage

**Details:** This techniques [54] defends against control-flow hijacking attacks in the kernel. These attacks are made possible by leverage kernel vulnerabilities such as buffer-overflows, or use-after-free vulnerabilities. Furthermore, this technique defends against kernel rootkits, which inject and execute kernel-level code. This technique also defends against data leakage attacks. This is significant as information leaked by the kernel about the layout of kernel memory could be used to mount a control injection attack.

**Description:**

**Details:** This technique randomizes both the code and the data within the operating system, and supports re-randomization during runtime. This re-randomization is supported through an LLVM link-time transformation that embeds relocation and type information into the final process binary. This allows for several distinct classes of runtime re-randomization. The code is re-randomized by shuffling the symbol table. To add entropy to the code re-randomization, dummy padding functions can be added to the symbol table (this negligibly affects the memory footprint of the task due to demand paging). Entropy is added to the function layout by adding a dummy basic block to the beginning of each function, a technique the authors call *basic-block shifting*. Static data objects are also re-randomized at runtime. Similar code re-randomization techniques are also applied here, in particular randomizing the static and read-only data in the symbol table. The layout of structs can also be re-randomized, with some limitations (*e.g.*, unions). The runtime stack is also re-randomized to change the base address, as well as the relative offset of objects on the stack by adding a stack-padding strategy. By adding a wrapper to `malloc/mmap`-like calls, random padding can be added around dynamically allocated objects, and these objects can also be permuted. Finally, loadable kernel modules can also be re-randomized. However, to support this functionality, dynamic linking within the operating system is forbidden.

**Entities Protected:** The operating system in which this technique is applied.

**Deployment:** This technique could theoretically be applied in any operating system. However, operating system fine-grained randomization is much easier to implement and more efficient in a microkernel-based operating system. Randomizing a monolithic kernel such as Linux would require significantly more effort.

**Execution Overhead:**

- The authors report experimental results that show approximately 1% overhead on average for the SPEC CPU 2006 benchmarks (which were ported to their prototype system).

- The worst-case overhead occurred for the perlbench program, which exhibited a 36% overhead. This is attributed to the high number of dynamic memory allocations.
- The runtime overhead of the system is a function of the randomization frequency. If randomization is applied every 1s, the overhead is over 40%.

**Memory Overhead:**

- The authors report experimental results that show approximately 15% increase in the size of the memory state.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☒ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** As mentioned previously, to support rerandomizing loadable kernel modules, dynamic linking is disallowed in the kernel.

**Weaknesses:** This technique has a few weaknesses. First, this technique is realistically only tenable in a microkernel. This limits its applicability, as in practice, major operating systems like Linux have monolithic kernels, and these techniques would be difficult if not impossible to apply to a monolithic kernel.

The overhead of re-randomization is strongly tied to the re-randomization frequency, and thus in order for the overhead to be tenable, the randomization interval must be several seconds or more. Such a long re-randomization interval may be long enough for an advanced attacker to launch an attack.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique significantly increases the effort of an attacker to compromise the kernel and carry out a code- or control-flow hijacking attack. The attacker would need to compromise the system within a short time interval before re-randomization is applied. This may limit the complexity or effectiveness of the attacks that could be carried out.

**Availability:** No code publicly available.

**Additional Considerations:** The current prototype is implemented in the MINIX 3 microkernel, and not easily portable to other operating systems.

**Proposed Research:** The authors note that the link-time modifications they make to enable re-randomization at runtime could be applied in other domains, such as in userspace applications.

**Funding:** European Research Council

### 4.1.5 FUNCTION-POINTER ENCRYPTION

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** This technique [55] defends against control injection through indirect buffer-overflow attacks on the heap by encrypting all function pointers so they cannot be modified.

**Description:**

**Details:** This technique aims to prevent indirect buffer-overflow attacks by making it difficult for the attacker to overwrite a function pointer with a chosen value. The GNU Compiler Collection (GCC) is patched so that at link/load time all function pointers `*fp` are replaced by `*fp` XOR `address(fp)` XOR `rand` where `rand` is a 32-bit random number, chosen at the start of execution. Using `rand` provides a high degree of unpredictability if the attacker does not know it, and it is chosen independently at the start of every execution so it should be difficult to guess. Incorporating `address(fp)` makes two different pointers to the same function have different keys. Additionally, this makes it so that the attacker cannot learn an encrypted value for one pointer and substitute it for another, changing the location of the original pointer. The "key" is effectively `address(fp)` XOR `rand` and is used symmetrically to decrypt the respective function pointer when it is dereferenced. If an attacker manages to find a buffer overflow vulnerability and exploit it to overwrite a function pointer, he or she will not be able to forge an encrypted address that will point to his or her chosen location when it is decrypted (since the attacker does not know `rand`).

**Entities Protected:** All programs running on a machine utilizing this technique.

**Deployment:** This technique could be applied to any generic machine by modifying the compiler and operating system.

**Execution Overhead:**

- The authors show an experimental slowdown of approximately 4%.

**Memory Overhead:**

- The size of the executable in memory is increased by addition of the encryption/decryption keys. The paper does not measure this effect but it is likely small (each function pointer approximately doubles in size)

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** None

**Weaknesses:** This technique is vulnerable to an attacker that has a copy of the program and can learn the encrypted value of a function pointer at runtime (through violation of memory secrecy). The function pointer is masked by its own address, which can be determined by an attacker running a copy of the program, and a random key, which can be deduced if the encrypted function pointer is known along with the unencrypted function pointer and its address (since the encryption function is just XOR). This effectively recovers the secret encryption key and would allow an attacker to forge a pointer to any chosen location that would work for *any* function pointer in the program (not just the one that the attacker originally learned). Techniques exist that would allow an attacker to exploit a vulnerable program to obtain one or more encrypted function pointers.

The above threat can be partially mitigated by using a cryptographic hash function instead of XOR when combining `rand` and `address(fp)`. This would still allow an attacker to forge the specific function pointer that was leaked to him, but it would not make other unrelated function pointers vulnerable (since the hash cannot be reversed and `rand` is not learned). Load time would be significantly slower while the linker computes hashes for each function pointer, but runtime would be the same because encryption and decryption would still be XOR (just the calculation of the individual keys changes). Full mitigation of this threat requires use of an encryption function that is secure against a known plaintext/ciphertext attack.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** Makes it difficult for an attacker to redirect program control to a chosen address unless he or she can both obtain a copy of the program executable and violate memory secrecy to obtain encrypted addresses.

**Availability:** This technique is used in several Linux distributions that we know about. Fedora Core encrypts function pointers in libc but not in other programs or libraries. Red Hat Enterprise has a reference to encrypting function pointers in one of its whitepapers but it is unclear what the scope of it is in their implementation.

**Additional Considerations:** This technique is also like ASLR. It has no significant downside, so if it is available, it is advisable to use it even if it has weaknesses.

**Proposed Research:** In the original paper, XOR was chosen as an encryption function because it is very fast and causes little overhead in the program execution. Using a secure encryption function at the time was not possible. Since publication, Intel has added a hardware instruction set for AES that can encrypt/decrypt in a small number of cycles. We propose that this scheme be implemented with XOR replaced by AES encryption/decryption done in hardware in order to evaluate the effect

on performance. Such a scheme would be secure against an attacker with any knowledge of the program except the encryption key. The question of where to store the encryption key is still open, but it should be possible to store it such that it would require an additional exploit in the kernel to bypass. It may also be possible to extend this technique to direct buffer overflow attacks (overwriting stack return addresses) but the implementation would be considerably different.

Barring AES encryption, this technique could also be made more robust by combining it with some kind of memory randomization. The most straightforward method would be to choose one that is implemented as a binary rewriter; from the point of view of the loader which does the encryption it would be no different but the executable on each machine would be randomized differently, making it much more difficult for an attacker (see above attack requirements).

**Funding:** National Science Foundation, Air Force Research Laboratory

### 4.1.6 CODE SHREDDING

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** This technique [56] defends against exploits based on memory-corruption vulnerabilities in which the attacker is able to overwrite all or part of a code pointer to an arbitrary value. The attacker is also assumed to be able to disclose a valid memory address within the program code. This technique works in conjunction with Data Execution Prevention (DEP), which prevents the attacker from modifying code.

**Description:**

**Details:** This technique effectively randomizes the address space of the code region at the byte level. Code addresses are *self validating* by encoding a checksum of the address in some high-order bits. A technique called *code mirroring* is used to replicate the binary for each unique checksum value. Each replica of the binary is referred to as a *segment*. At each control-flow transfer instruction, the target address is validated by checking the checksum value in the high-order bits of the address, and thereby validating the address is in the correct segment.

**Entities Protected:** All programs running on a machine utilizing this technique.

**Deployment:** This technique can be deployed on any generic machine by developing a process-level *virtual machine monitor (VMM)* that supports dynamic binary instrumentation to support the validation of addresses. This technique could also be realized at the OS or hardware level.

**Execution Overhead:**

- The authors show experimentally a slowdown factor of 3.65x to 26.1x.

**Memory Overhead:**

- Code mirroring increases the size in virtual memory by a factor of $2^c$ where $c$ is the number of bits in the checksum.
- Code mirroring can be realized by mapping one physical page into multiple virtual addresses. Consequently, the physical memory overhead is significantly less than the virtual memory overhead.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** Code shredding is a form of ASLR, and therefore has similar interdependencies.

**Weaknesses:** This technique introduces a significant amount of overhead into the system, which renders it impractical for many systems. The technique as implemented by the authors also is subject to several other limitation. The technique cannot be applied to dynamically loaded libraries, which often contain a significant portion of the executed instructions in an application. The technique is not applied to the Code Shredding module itself that implements the proposed technique. Therefore, the Code Shredding module itself is a valid target for a control-flow hijacking attack. This technique also does not apply to dynamically generated code, such as JIT code. Therefore, it is also vulnerable to JIT spraying attacks. Finally, the existing implementation cannot be applied to all binaries–some result in crashes for unknown reasons.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** Increases the level of effort for attackers in many circumstances. Since some instruction sequences cannot be processed by this technique, portions of executables may remain vulnerable.

**Availability:** No code publicly available.

**Additional Considerations:** The significant overhead introduced by Code Shredding as well as the weaknesses discussed render it impractical for operational deployment.

**Proposed Research:** Implementing similar functionality in the OS or hardware so the Code Shredding module itself cannot be the target of a control-flow hijacking attack.

**Funding:** Unknown

### 4.1.7 BINARY STIRRING

**Defense Category:** Dynamic Runtime Environments

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** This technique [57] defends against control-flow hijacking attacks through gadgets, or short instruction sequences, that are contained within the main application binary itself, instead of libraries. To carry out such an attack, an attacker would need to exploit a memory-corruption vulnerability to redirect control flow. Without this technique, the addresses of gadgets within the application binary are static, and can therefore be easily discovered by an attacker on a remote machine with a copy of the binary.

**Description:**

**Details:** The goal of this technique is to randomize the address space of the main application binary for each individual invocation of the application. This is accomplished via two steps. First, a binary-rewriting tool is used to analyze and rewrite an application binary to add a load-time *self-transforming instruction relocation* (*STIR*) phase. Second, the runtime phase randomizes the address space based on metadata from the analysis conducted in the binary-rewriting tool that identified contiguous blocks of code.

**Entities Protected:** Any binary can be written using this technique's binary-rewriting tool. Any application processed with this tool is protected by this technique.

**Deployment:** This technique can be applied to applications on any generic machine by processing their binaries with this tool. Application distributors can process applications before distributing them to others so that end users are seamlessly protected.

**Execution Overhead:**

- The authors show experimentally an average slowdown of 1.6%

**Memory Overhead:**

- The authors show experimentally that the file size and code-section size of each binary increases on average by 73% and 3%, respectively.
- The authors show experimentally an average increase in the process memory size of 37%.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☒ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique randomizes the application binary. For increased protection, it can be used in tandem with other ASLR techniques that randomize the library locations.

**Weaknesses:** The main weakness of this defense is that all code addresses are static after initialization. Therefore, it is vulnerable in the presence of a memory-disclosure vulnerability that could leak code addresses after the initial randomization. At attacker can use such information to launch a tailored control-hijacking attack. This attack technique was later formalized as JIT-ROP [58].

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** This technique increases the level of effort for attackers by forcing the attacker to do reconnaissance for every program invocation to identify the locations of any gadgets they wish to leverage in a control-flow hijacking attack.

**Availability:** No code publicly available. A commercial product from Polyverse [59] appears to leverage a similar technique.

**Additional Considerations:** For this technique to be maximally effective, it must be applied to every application running on a machine. However, it may be possible, in environments like Linux, to apply such randomization to all package in a centralized package manager. This has the benefit that end users do not need to process all of their applications to get the desired protection.

**Proposed Research:** Only code available at load time is randomized. Notably, just-in-time-compiled code is not affected by this technique. The authors also propose considering the possibility of rerandomizing the address space periodically during execution to provide stronger security.

**Funding:** Air Force Office of Scientific Research, National Science Foundation, Defense Advanced Research Projects Agency

### 4.1.8 INSTRUCTION LAYOUT RANDOMIZATION

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [60], Instruction Layout Randomization (ILR), defends against code-reuse attacks that hijack and redirect control flow through gadgets in the code. In such attacks, the attacker must know where to redirect control flow to achieve the desired effects. The attacker does not have direct access to the system or protected program, but is assumed to have access to a copy of the executable for offline analysis.

**Description:**

**Details:** More coarse-grained address-space randomization techniques are vulnerable if code addresses are disclosed and the attacker can then infer the location of other useful code or gadgets. This technique applies fine-grained address-space randomization at the granularity of a single instruction in order to make identifying and redirecting control flow through specific gadgets more difficult. ILR includes two key components, the first of which is an offline analysis that is used to randomize the instructions and include relevant metadata used at runtime. The second key component of ILR is a *process-level virtual machine* (*PVM*) that fetches the randomized instructions and executes them in the correct order.

**Entities Protected:** Every process that has been processed by ILR and run with the associated PVM.

**Deployment:** This technique could be applied to any generic machine by modifying the compiler.

**Execution Overhead:**

- Experimental results show an average execution-time increase of approximately 15%. In some cases, the execution overhead can be much higher, nearly 2x.
- The PVM on which ILR is built incurs an overhead of approximately 8%.

**Memory Overhead:**

- On-disk memory increase as high as 264 MB.
- In-memory overhead as high as 345 MB.
- Authors note that their prototype has not been optimized for memory usage and conjecture the memory use could be substantially reduced.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** ILR is built upon a PVM, similar to some other techniques, such as described in Section 4.2.2. To combine such techniques, the PVM would have to be extended to support both feature sets concurrently, assuming they are not mutually exclusive.

**Weaknesses:** After the instructions are randomized, their addresses may still be vulnerable to memory-disclosure vulnerabilities. If an attacker can identify the address of a function or gadget, they may still be able leverage the code in a control-flow hijacking attack. Additionally, the PVM is not itself randomized as the rest of the application is. Therefore, the PVM itself is a potential vulnerability.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** ILR significantly increases the level of effort for attackers by randomizing the location of every instruction in the program. The attacker therefore cannot leverage the context of nearby code to infer the control flow to subsequent instructions that could be leveraged in a control-flow hijacking attack.

**Availability:** No code publicly available.

**Additional Considerations:** ILR does not randomize the location of shared libraries. Libraries are commonly the target of control-flow hijacking attacks. Also, there are some corner cases and compiler optimizations that ILR cannot handle.

In more recent work [61], Kim et al. considered ILR hardware acceleration. They proposed integrating ILR into the processing pipeline, and maintaining two program counters: one for the randomized instructions and one for unrandomized instructions. In this way, the unrandomized addresses can be used to allow better cache utilization, while randomized addresses are used to provide better security. Their simulation results suggest a minimal increase in the power consumption of a processor implemented with this functionality (<1%), but 1.63x better performance, as compared to a more naïve ILR software-based implementation.

**Proposed Research:** The authors propose optimizing their implementation to improve the memory overhead.

**Funding:** National Science Foundation, Army Research Office, Air Force Research Laboratory, and Air Force Office of Scientific Research

### 4.1.9 IN-PLACE CODE RANDOMIZATION

**Defense Category:** Dynamic Runtime Environments

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [62] defends against code reuse and ROP attacks. To carry out such attacks, an attacker must know the addresses to which to divert control flow. The attacker can provide input to the vulnerable software, either directly over the network, or through a file or other program input. For example, the authors consider malicious PDF inputs to Adobe Reader. This technique defends against ROP attacks through gadgets, or short sequences of a few instructions, instead of whole functions.

**Description:**

**Details:** The technique is an amalgamation of several binary-modification techniques that modify the instructions in the program but not the logical functionality of the program. By replacing instructions or instruction sequences with different but logically equivalent instructions, the gadgets in the program are either removed, or replaced by a different gadget, which may have different, unexpected behavior when used in a ROP attack. In carrying out these binary-modification techniques, the length of the code is unchanged such that all existing branch-target addresses are unchanged. One such binary-modification technique is to substitute instructions one for one, where the instructions have the same functionality and length (x86 instructions in general are variable length). For example, addition can be replaced with negative subtraction. The next binary-modification technique is instruction reordering. When a compiler generates a binary, it orders instruction in such a way to optimize for the execution time of the application based on the microarchitecture of the processor. There are many logically equivalent ways to order instructions, and by reordering instructions gadgets can be altered, moved, or eliminated. Another binary-modification technique is to modify the registers that are used. There are several general-purpose registers, and so by permuting which registers hold what data, the set of gadgets in the application are modified, making it more difficult to launch a deterministic ROP attack. These techniques can all be applied without the source code or any debugging symbols.

**Entities Protected:** Any applications that have been processed by these binary-modification techniques.

**Deployment:** This technique could be applied to any generic machine by processing any applications to be protected with a similar tool.

**Execution Overhead:**

- Negligible execution-time overhead. The number of instructions executed is the same, but the instruction order may be slightly slower given the processor pipeline.

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

- ☐ Data
- ☐ Source Code
- ☐ Compiler/Linker
- ☐ Operating System
- ☐ Hardware
- ☐ Infrastructure

(No modification required)

**Expertise Required to Implement:**

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☐ Custom Programmer (General Knowledge)
- ☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

- ☒ Seamless
- ☐ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique only randomizes instructions within basic blocks. To increase its effectiveness, it should be used in combination with other ASLR techniques that randomize the layout of the basic blocks within the address space.

**Weaknesses:** There are limitations to how much diversification can be achieved by randomizing the binary in place. The experimental results show that this technique can modify, eliminate or break approximately 75% of the gadgets in a binary. However, this demonstrates that some gadgets still remain. Furthermore, the instructions are never re-randomized, and therefore are vulnerable to memory disclosure.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique makes it more difficult to carry out ROP attacks by altering the instructions and their locations to break the gadgets upon which the attacker relies. Some gadgets remain, and could be used by a more determined attacker to carry out an attack. Alternatively, if an attacker can disclose the locations of gadgets that exist after in-place randomization is applied, the attacker can still carry out a ROP attack.

**Availability:** Source code is publicly available at `http://nsl.cs.columbia.edu/projects/orp/`.

**Additional Considerations:** This defense does not defend against JIT-ROP [58] attacks, as an attacker can disclose the gadgets after in-place randomization.

**Proposed Research:** The authors propose to extend the coverage of randomization through more advanced data-flow-analysis methods. Furthermore, the tool was developed specifically for 32-bit x86 PE executables, and the authors would like to support ELF and 64-bit executables.

**Funding:** Defense Advanced Research Projects Agency, Air Force Research Laboratory, European Commision

#### 4.1.10 MORPHISEC

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [63] defends against control-flow hijacking attacks in userspace application. In particular, the technique defends against attacks that exploit *a priori* knowledge of the layout of memory.

**Description:**

**Details:** Morphisec is a commercial product and therefore technical details are not readily available. From the marketing literature, the product randomizes the layout of memory, including libraries. However, the granularity of randomization is unclear. Morphisec also maintains a copy of the original memory layout. Upon malicious input to conduct code or control injection, the randomized application will crash. Furthermore, the malicious input can be fed, presumably in an isolated environment, to the original memory layout to conduct a forensic analysis. Finally, the results of this forensic analysis can be forwarded over the network to a dashboard process where an administrator can monitor the health of the enterprise.

**Entities Protected:** All applications that are processed by the product.

**Deployment:** There is not publicly available information on the systems on which this technology can be deployed.

**Execution Overhead:**

- No publicly available information.

**Memory Overhead:**

- No publicly available information.

**Network Overhead:**

- No publicly available information.

**Hardware Cost:**

- No publicly available information.

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification required)

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** The Morphisec marketing materials claim that the product complements and cooperates with other security agents.

**Weaknesses:** This defense does not defend against JIT-ROP [58] attacks, which disclose the current location of gadgets in memory, and use that information to construct a malicious payload on the fly.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** This technique increases the difficulty of carrying out many types of attacks. Additionally, the forensic analysis made possible by this technique could be used to identify commonly exploited vulnerabilities, as well as the actors leveraging these attacks.

**Availability:** This is a commercial product, and demos and licensing are available for current deployment.

**Additional Considerations:** None

**Proposed Research:** This technique would be more effective if it supported runtime re-randomization.

**Funding:** Investors and customers

#### 4.1.11 OXYMORON

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** This technique [64] defends against control injection attacks. The specific threat model considered is that of JIT-ROP [58], in which it is assumed the attacker can disclose and/or corrupt arbitrary memory. This allows the attacker to disclose the location of gadgets within a program, even if they have been randomized at load time, and construct a ROP attack at runtime that can be launched via a memory-corruption vulnerability.

**Description:**

**Details:** This technique applies ASLR at the page granularity. In order to prevent the disclosure of code pointers within the executable, an additional level of indirection is added called the *Randomization-agnostic Translation Table* (*RaTTle*), which maps an offset in the table to the destination address. The RaTTle is stored in a different x86 memory segment, so that it is not subject to memory disclosure. Oxymoron is developed as a static translation tool that will convert existing references through the RaTTle.

**Entities Protected:** All programs that are converted using the Oxymoron tool.

**Deployment:** This technique relies on hardware support for memory segmentation, as is available on x86. The technique can be used on any machine supporting this functionality.

**Execution Overhead:**

- Across all SPEC 2006 benchmarks, the authors observed an average overhead of 2.7%.

**Memory Overhead:**

- To support randomization at the page level, each page is a section in the ELF file, resulting in 1.76% overhead in the file size.
- The additional instructions needed to jump through the RaTTle result in 12% overhead in ELF file size for the SPEC 2006 benchmarks.
- The RaTTle consumes on average 19% additional memory in the code segment in the SPEC 2006 benchmarks.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification required, as implemented by the authors. However, the technique could alternatively be implemented in the compiler or the loader.)

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** None

**Weaknesses:** This technique has been shown vulnerable [65], as an attacker can still disclose code pointers from the stack and/or the heap. A heap-allocated C++ object has a *vtable*, which contains code pointers that are not redirected through the RaTTle that can be disclosed. It has been shown that enough code pointers can be harvested through the stack and the heap to construct a realistic exploit [65]. Furthermore, this technique does not protect code pages that are dynamically generated, for example, as in JavaScript run in the browser.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique makes disclosing the addresses of useful gadgets more difficult. However, as discussed above, it is still feasible to disclose enough useful gadgets to construct a malicious payload via the stack and the heap.

**Availability:** Not publicly available

**Additional Considerations:** None

**Proposed Research:** In order to provide better security, this technique would need to defend against the enhanced JIT-ROP attack presented in [65]. Furthermore, it would be valuable to randomize dynamically generated code pages.

**Funding:** Unknown

### 4.1.12 DYNAGUARD

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** Many code- and control-injection attacks begin with a stack-buffer overflow, which allows the attacker to corrupt the stack to inject code or redirect control flow, a technique often called stack smashing. To mitigate such attacks, stack canaries, or a random integer chosen at program start time, is written to the stack before each return address. A stack overflow will overwrite the canary before overwriting subsequent return addresses. The canary is checked before returning to ensure that the return address has not been corrupted.

A recent attack technique called blind ROP (BROP) [66] has demonstrated how the stack canary can be disclosed remotely via brute force on a service that restarts after each crash. This technique [67] defends against BROP-like attacks that use brute force to disclose the stack canary, which exploit the fact that existing stack-canary implementations, such as implemented in GCC, use when a process forks, the parent and child process retain the same canary value.

**Description:**

**Details:** This technique changes the stack canary of the child process when it is forked. This requires changing the canary value in the thread local storage (TLS), as well as within each frame on the stack. To do so, each canary address is stored in a canary address buffer (CAB). When a process forks, before the child process begins executing, every canary on the stack must be updated to a new random value. This functionality can be added either at compile time, or through dynamic binary instrumentation. The former approach is significantly more efficient, whereas the latter can be applied without source code.

**Entities Protected:** All processes in which the technique has been applied.

**Deployment:** This technique could be employed on any generic machine.

**Execution Overhead:**

- The authors report a measured increase in the execution time of programs of 1.2% when the technique is applied at compile time.
- The authors report a measured increase in the execution time of programs of 70% when the technique is applied via binary instrumentation. However, compared to the baseline (binary instrumentation, but no defense), the overhead is only 2.92%.

**Memory Overhead:**

- Minimal memory overhead. The code section is increased minimally by adding a few extra instructions to log the canary address in the CAB. The CAB is also a small buffer, with one word per stack frame.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☒ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

- ☒ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** This technique complements control-flow-integrity defenses, which check the execution follows valid control flow. Furthermore, the version of this technique implemented via dynamic binary instrumentation could conceivably be integrated with other defenses based on dynamic binary instrumentation.

**Weaknesses:** Stack canaries are a simple, low-overhead way of increasing the burden on the attacker. This technique makes stack canaries more resilient to brute-force attacks against the stack canary such as BROP. However, there are many other ways a process can be exploited even in the presence of this technique. For example, a memory-disclosure vulnerability can be used to disclose the canary value, which allows the attacker to directly bypass this defense.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** This technique inhibits the ability of the attacker to disclose the stack-canary value via brute-force in applications that fork new processes after each crash.

**Availability:** Source code is publicly available at `https://github.com/nettrino/DynaGuard`.

**Additional Considerations:** While the authors have presented two means of implementing this technique (via the compiler and dynamic binary instrumentation), in cases where the source code is available, it is preferable to deploy this technique via a compiler plugin.

**Proposed Research:** A potential avenue of future work would be to re-randomize the canary value dynamically during the program's execution. This would help defend against stack-smashing attacks where the canary value has been gleaned by the attacker through a memory-disclosure vulnerability. Given that the CAB stores the address of every canary value, implementing such functionality would only require hooks at the relevant re-randomization points.

**Funding:** Office of Naval Research

#### 4.1.13 SHUFFLER: FAST AND DEPLOYABLE CONTINUOUS CODE RE-RANDOMIZATION

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** This technique [68] defends against control injection attacks. The specific threat model considered is that of code reuse relying on memory disclosure. Code reuse, such as ROP, requires the attacker to know where in memory gadgets are located. ASLR and other memory-randomization techniques prevent naive ROP. However, memory-disclosure vulnerabilities can leak addresses to the attacker, effectively de-randomizing memory and allowing ASLR to be bypassed. Shuffler mitigates such attacks by periodically re-randomizing memory, which makes leaked addresses no longer useful for de-randomization after a small time window.

**Description:**

**Details:** Shuffler re-randomizes a process' address space periodically, with a recommended period of 50ms. This period is posited by the authors to be too rapid for an attacker to leak and utilize a memory address while being infrequent enough to minimize performance degradation. It re-randomzies at the granularity of individual functions. These are placed randomly throughout the memory space by creating a permuted copy of the current memory layout concurrent with program execution. A lookup table is used to track code pointers and update their value to the newly permuted function location. Return addresses are handled differently, by XOR-ing each with a per-thread key. This key is changed on every re-randomization. The authors note that using a lookup-table approach for return addresses would induce substantial overhead due to their dynamic nature. Once shuffling is complete, the execution is switched to the permuted copy and the original layout is unmapped. In order to track code pointers, Shuffler requires binaries that retain symbol and relocation data, but does not require access to source code.

**Entities Protected:** All user-space programs that are converted using the Shuffler tool.

**Deployment:** This technique requires a patch to the program loader in order to install Shuffler threads inside running binaries.

**Execution Overhead:**

- Across all SPEC 2006 benchmarks, the authors observed an average overhead of 14.9% when using a re-randomization period of 50 ms. This assumes a spare CPU core is available for asynchronous shuffling computations, however.

**Memory Overhead:**

- Shuffler maintains an in-flight copy of code sections, which incurs a 100% increase in memory usage.

- Additionally, Shuffler itself incurs a 250 KB overhead per process.
- Shuffler also requires metadata on functions, symbols, and relocations, which varies per program but is on the order of 30 MB-100 MB.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

- ☒ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** None

**Weaknesses:** This technique re-randomizes periodically and at the granularity of functions. Attackers may be able to take advantage of both of these properties. If a memory-disclosure vulnerability can be triggered and a payload launched within the re-randomization period, then the attack will succeed due to Shuffler acting too slowly to invalidate the data contained in the memory disclosure. Additionally, if an attacker can find gadgets within the function pointed to by a target pointer, those gadgets could still be used, as their locations relative to one another remain constant across re-randomizations.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** This technique makes disclosing the addresses of useful gadgets more difficult. However, as discussed above, it does not provide complete protection

**Availability:** Not publicly available

**Additional Considerations:** None

**Proposed Research:** This technique would benefit from finding a way to synchronize re-randomization with attacker actions rather than relying on a fixed time period. In addition, re-randomizing on the basic-block level rather than the function level would prevent attackers from being able to use intra-function gadgets.

**Funding:** Office of Naval Research and National Science Foundation

#### 4.1.14 RUNTIME ASLR (RASLR)

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** Runtime ASLR (RASLR) [69] defends against control-injection attacks. The specific threat model considered is that of clone-probing attacks against daemonized servers. These attacks rely on the fact that the child processes of daemonized servers (*e.g.*, Apache, Nginx, OpenSSH) share the memory layout of the parent process. This allows attackers to incrementally guess the value of pointers in memory and brute force ASLR after at most 1024 guesses (for 64-bit systems). A failed guess causes the process to crash, but the daemon will instantiate a new process with the same layout as the old one, allowing the attacker to gradually de-randomize memory. This technique re-randomizes the new process created by every `fork` system call, preventing attackers from using clone probing attacks.

**Description:**

**Details:** RASLR re-randomizes the address space of a spawned child process at `fork`-time, ensuring that every child has a different layout from each other and from their parent. The granularity is at the module-level, that is, each executable or library is treated as a contiguous whole whose base address is randomized. RASLR relies on dynamic taint tracking at run time to identify pointers whose value must be updated after re-randomization. This process has no false negatives (missed pointers that are not updated), and a near-negligible low false negative rate (non-pointers are updated). It requires no access to source code or debugging symbols, and will operate on arbitrary binaries.

**Entities Protected:** All user-space programs that operate as daemons and spawn sub-processes to handle sessions or requests

**Deployment:** This technique requires a patch to the program loader.

**Execution Overhead:**

- RASLR must conduct a one-time operation at program load time that can add 10 seconds to its startup time. After this, there is negligible overhead in the daemon process.
- Every time a child process is started, RASLR must re-randomize its memory layout. This delays the sub-process startup time by less than 150 ms.

**Memory Overhead:**

- The authors did not report on memory overhead. It is likely minimal, as the randomizer only exists in process memory at load time, and removes itself prior to execution.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** RASLR currently relies on the Pin dynamic analysis framework. In general, some system for attaching and detaching from a process is required.

**Weaknesses:** This technique is not effective against attacks relying on memory disclosures (either direct or indirect). It only applies to brute-force attacks that exploit predictable memory layouts in child processes. Additionally, if an attacker can find gadgets within the module pointed to by a target pointer, those gadgets could still be used, as their locations relative to one another remain constant across re-randomizations.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique makes brute-forcing of ASLR-protected daemons infeasible. However, as discussed above, there are many other ways to attack such systems (*e.g.*, memory disclosures)

**Availability:** Not publicly available

**Additional Considerations:** None

**Proposed Research:** This technique would benefit from finer-granularity randomization to disrupt use of gadgets with known relative offsets from one another. Additionally, re-randomization during the lifetime of sub-processes could mitigate memory disclosures.

**Funding:** Office of Naval Research, Department of Homeland Security, United States Air Force, and National Science Foundation

#### 4.1.15 LEAKAGE-RESILIENT LAYOUT RANDOMIZATION FOR MOBILE DEVICES

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** LR$^2$ [70] defends against control-injection attacks that rely on memory disclosures. ASLR and other memory randomization techniques prevent naïve ROP. However, memory-disclosure vulnerabilities can leak addresses to the attacker, effectively de-randomizing memory and allowing ASLR to be bypassed. The specific threat model considered is an attacker using memory disclosures to conduct ROP attacks on mobile devices, whose hardware does not fully support protection of virtual memory. This makes many disclosure defenses, such as hardware-backed execute-only memory (XoM), infeasible or impossible to deploy.

**Description:**

**Details:** LR$^2$ uses three techniques to mitigate memory disclosures: software-based XoM, forward pointer hiding, and return pointer encryption. The first technique splits the address space in two and relegates one half for code and one for data. This creates a single bit in the address that denotes code vs data. By masking that address on `load` instructions (*i.e.*, always testing that it points to data), attackers trying to directly read code in order to find gadgets will cause a program crash. This mitigates direct memory disclosures.

Forward pointer hiding replaces all function pointers to code with pointers to a randomized trampoline table. When a pointer is dereferenced, code stubs in the table cause the original function to be executed. This prevents attackers from learning where in code a pointer is pointing. Return-address encryption has a similar effect, but instead of a lookup table, it uses XOR encryption with a per-function key stored in XoM. These techniques mitigate indirect memory disclosures.

**Entities Protected:** All user-space programs that are converted using the LR$^2$ tool.

**Deployment:** Currently, LR$^2$ requires recompilation of source code in order to effect its program transformations. The authors note that this is not a fundamental requirement, however, and could in principle be conducted on binaries via rewriting.

**Execution Overhead:**

- Across all SPEC 2006 benchmarks, the authors observed an average overhead of 6.6%.

**Memory Overhead:**

- The authors reported a memory overhead of 5.6%.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** None

**Weaknesses:** The use of function-pointer trampolines has been successfully bypassed via profiling of memory layout during execution [71]. These attacks rely on predictable stack and heap data layouts to identify where the program is in execution, and infer based on this what functions trampolines are pointing to. In addition, return pointers are protected by an XOR-based encryption scheme whose key never changes. This could allow an attacker to eventually infer the key value by repeated observation, as XOR is only secure when used as a one-time pad.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique makes disclosing the addresses of useful gadgets more difficult. However, as discussed above, it does not provide complete protection.

**Availability:** Not publicly available

**Additional Considerations:** None

**Proposed Research:** This technique would benefit from finding a way to efficiently re-randomize the keys used to encrypt return pointers. Deployment would be made much simpler if the program transformations could be made to a binary rather than source code. Finally, applying this technique to Just-In-Time compilation (such as is used in modern browsers) could substantially increase its coverage against attack vectors.

**Funding:** Defense Advanced Research Projects Agency and National Science Foundation

#### 4.1.16 BLENDER: SELF-RANDOMIZING ADDRESS SPACE LAYOUT FOR ANDROID APPS

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** Blender [72] defends against control-injection attacks on Android applications. ASLR on Android is much weaker than on general purpose computers due to optimizations made to increase app performance. These optimizations place pre-compiled code (Android apps are stored as bytecode) and shared system libraries in predictable locations. Part of this code includes high-level framework APIs compiled into native code, which makes powerful gadgets available to attackers. For example, these include sending an SMS or getting GPS coordinates. Blender randomizes these code regions on app start-up, restoring the guarantees of ASLR to the protected app.

**Description:**

**Details:** Blender consists of two primary components, each responsible for randomizing a memory region whose location is made predictable by system optimizations. The first, BlenderLRM, randomizes the location of system libraries. These libraries are pre-loaded in the address space due to optimizations, and are dynamically linked to other libraries whose dependencies have already been resolved at load-time. In order to randomize system library base addresses without breaking these dependencies, BlenderLRM computes a dependency graph for each library and uses this to update offset tables in its dependencies.

The second component, BlenderART, is responsible for randomizing the Android Runtime (ART). This code region contains pre-compiled API functions that implement high-level operations (e.g. sending an SMS) in native code usable by an attacker. Its memory location is fixed and cannot normally be randomized due to the use of absolute addresses. BlenderART uses binary rewriting to patch all absolute address references with new values after randomization.

**Entities Protected:** All user-space Android applications that are converted using the Blender tool.

**Deployment:** Blender requires application developers to include it in their source code in order to protect that app.

**Execution Overhead:**

- Blender performs a one-time randomization that increases app startup time by approximately 300 ms.

**Memory Overhead:**

- Blender incurs a constant overhead of 6 MB.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** None

**Weaknesses:** Blender suffers from the weaknesses of traditional ASLR. A single memory disclosure vulnerability can allow attackers to de-randomize memory and execute a ROP attack. Additionally, entropy-exhaustion or brute-force attacks may be possible. The authors do not perform a comprehensive analysis of how many bits of entropy are added by their defense.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique restores the standard guarantees of ASLR to Android applications, increasing the difficulty of launching code-reuse attacks. However, as discussed above, it suffers from the well-known weaknesses of ASLR.

**Availability:** Not publicly available

**Additional Considerations:** None

**Proposed Research:** This technique would benefit from finding a way to efficiently re-randomize memory in order to mitigate memory disclosures. In addition, it currently has to be included by application developers on a per-app basis. A binary-only solution would enable users to selectively protect applications and make deployment much easier.

**Funding:** Unknown

#### 4.1.17 READACTOR

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** Readactor++ [73] and the Readactor [74] defense it builds upon defend against control injection attacks. Specifically, the threat model considers code-reuse attacks that rely on memory disclosure to bypass ASLR or other memory-randomization approaches. Two kinds of disclosure are considered. Direct memory disclosure is the leaking of the contents of code regions to an attacker. Indirect disclosure is the leaking or pointers into a code region, which attackers can both use directly, as well as use to potentially de-randomize that region. Readactor++ mitigates these attacks through a combination of execute-only memory (XoM) and fine-grained code diversity.

**Description:**

**Details:** Readactor++ relies on four broad techniques to defend different regions of memory, and has both compile-time and run-time components. The foundation of the defense relies upon implementing XoM (via extended page tables), and using this capability to render code regions unreadable. This alone prevents direct disclosure, as any attempt to leak code memory will cause an access violation.

To prevent indirect disclosure of pointers into code on the stack or heap, Readactor++ use code-pointer hiding. This technique replaces all pointers to code with "trampoline" pointers into an execute-only lookup table. When the trampoline is dereferenced, code in the lookup table invokes the original function. This means attackers cannot leak the location of the code region, only the location of the trampoline region, which does not contain usable gadgets.

Attackers could potentially still reuse code in execute-only regions by brute-forcing or guessing its location. To mitigate this, Readactor++ uses four compile-time code transformations to add fine-grained diversity to the program. Function permutation reorders the layout of functions in memory. NOP insertion probabilistically adds instructions that have no effect, but change the address of other instructions that might be used in gadgets. Register-allocation randomization changes the dataflow between registers. This disrupts the input and output registers of any gadgets used by the attack. Finally, stack-slot randomization permutes the stack location where saved register values are stored. These saved registers are often used as data inputs to gadgets, and randomizing them causes the wrong value to be placed in a register used by an attacker.

Finally, Readactor++ also protects sensitive tables of function pointers. These include virtual function tables in C++, and the Procedure Linkage Table (PLT) used by Linux processes. It uses a combination of execute-only memory and booby traps. First, the tables are split into a read-only and execute-only component, the latter of which contains sensitive code pointers. The table entries are permuted in a random order. This prevents direct disclosure. However, attackers could still try to randomly execute regions of these tables and observe the result. Due to their fairly small size,

this could be a source of indirect disclosure. To prevent this, Readactor++ adds booby-trap entries. These are never invoked by the legitimate program, but if an attacker tries randomly executing table regions, these will be triggered with high probability. A booby trap crashes the program and displays a warning that an attack was attempted.

Unlike many memory disclosure defenses, Readactor++ also protects Just-In-Time (JIT) compilers used by most modern web browsers. By its nature, JIT violates the usual rule that memory cannot be both writable and executable, which normally prevents code injection attacks.

**Entities Protected:** All user-space applications that are recompiled using Readactor++.

**Deployment:** Readactor++ requires a small kernel patch, as well as recompilation of source code on any protected application. It is currently only implemented on Linux.

**Execution Overhead:**

- When tested against the SPEC 2006 CPU Benchmark, Readactor++ had an average overhead of 8.4% and a worst-case overhead of 24%.

**Memory Overhead:**

- The authors did not provide an analysis. Trampoline tables would be the largest source of overhead, and would scale with the number of functions in a program.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** Readactor++ relies on Extended Page Tables to efficiently implement XoM. This is not available on all CPU platforms.

**Weaknesses:** Readactor++ attempts to avoid indirect memory leakage through the use of trampolines for code pointers. This approach, however, can by bypassed by profiling the data-memory layout of an application [71]. By inferring what function is pointed to by a trampoline pointer, attackers can still mount Return-into-Libc-like attacks. Additionally, execute-only protections can be bypassed by using DMA operations to access memory directly [71]. This permits direct memory disclosures and bypasses the technique. Finally, Readactor does not have a re-randomization component. This may enable attackers to brute-force applications that respawn on crash, especially if their memory layout is preserved (*e.g.*, server daemons).

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique makes code-reuse attacks relying on memory disclosures much more difficult. Attackers must find a way to either bypass the defense (*e.g.*, software-based DMA attacks) or infer the functions pointed to by trampolines, and subsequently overcome the dataflow randomization applied to registers.

**Availability:** Not publicly available

**Additional Considerations:** None

**Proposed Research:** This technique would benefit from finding a way to efficiently re-randomize memory in order to further mitigate memory disclosures. As discussed above, this makes applications that respawn on crash potentially vulnerable to brute-force attacks. In addition, extending Readactor++ to include IOMMU support would mitigate DMA-type attacks on memory permission.

**Funding:** Defense Advanced Research Projects Agency

#### 4.1.18 HEISENBYTE

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** Heisenbyte [75] defends against control-injection attacks. Specifically, the threat model considers code-reuse attacks that rely on memory disclosure to bypass ASLR or other memory randomization approaches. Rather than trying to prevent direct or indirect disclosure, Heisenbyte instead aims to make disclosed addresses impossible to actually exploit. That is, the gadgets found at an address, if used by the attacker, will cause the program to crash.

**Description:**

**Details:** Heisenbyte mitigates memory disclosures via destructive memory reads. In order to find gadgets, attackers must read executable memory, which is distinct on the instruction level from the CPU fetching bytes of code to execute. The defense takes advantage of this distinction. Whenever a user-space process reads executable memory, the bytes read are immediately randomized. This means that if an attacker attempts to use those bytes in a gadget, the program will crash as the random bytes are executed.

Heisenbyte does not randomize bytes if the kernel reads executable memory, as there are legitimate reasons for such reads. In addition, Heisenbyte must identify and, to the extent possible, relocate data that is intermixed with executable code memory pages in order to minimize overhead. This is achieved by a binary static analysis phase prior to execution of the program to be protected.

During runtime, Heisenbyte creates a copy of an executable code page the first time that page is read from. All read operations are resolved against the original page. However, each byte read in the original page is randomized in the copied page. All CPU fetch instructions (which are used to execute code) are resolved against the copied page. Thus, if executable memory is ever read, any subsequent attempt to run it will cause a program crash.

**Entities Protected:** User-space applications converted using Heisenbyte

**Deployment:** Heisenbyte requires relocation data to be available for binaries, and is currently implemented for Windows-based applications.

**Execution Overhead:**

- Across all SPEC 2006 benchmarks, the authors observed an average overhead of 18.3% and a worst-case overhead of 62%.

**Memory Overhead:**

- The authors reported an average overhead of 0.8%, and a worst-case of 10% on the SPEC benchmark.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** Heisenbyte's security guarantees require fine-grained ASLR to also be deployed.

**Weaknesses:** This technique does not defend against indirect memory disclosure via remote side-channels, which have been shown to be feasible attacks on real systems [76]. In addition, it is possible that attackers could infer adjacent code in memory without explicitly reading that code [77], within the bounds of ASLR granularity, by comparing the read code against a local copy of the binary. These adjacent gadgets would still be available, as their memory addresses were never explicitly read.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique makes usage (but not leakage) of memory disclosures more difficult. However, as discussed above, it does not provide complete protection.

**Availability:** Not publicly available

**Additional Considerations:** None

**Proposed Research:** This technique would benefit from finding a way to detect the use of remote side channels, or to invalidate their leaked data via, *e.g.*, re-randomization of executable memory. Additionally, randomizing adjacent code on the same memory page could mitigate inference attacks.

**Funding:** National Science Foundation

#### 4.1.19 STACKARMOR

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** StackArmor [78] is designed to protect against exploitation of spatial and temporal stack-based vulnerabilities. Spatial vulnerabilities correspond to corruption of memory outside the memory region intended for a buffer (*e.g.*, an overflow). Temporal vulnerabilities correspond to use of memory that is no longer valid, *e.g.*, uninitialized data or freed data. It does not consider a specific attack technique such as ROP, but aims to mitigate any kind of attack relying on stack vulnerabilities.

**Description:**

**Details:** StackArmor relies on static binary analysis and binary rewriting of a program in order to provide three defensive program transformations. First, all functions with pointers to local variables are identified, as these are not provably safe from spatial or temporal attacks. The stack frames of these functions will be placed at a random location rather than in contiguous stack memory, and will be isolated (surrounded by unmapped memory) from other data. Next, all local variables that are not provably initialized will be initialized to 0. This prevents temporal vulnerabilities arising from use of uninitialized data. Finally, any buffer that can be safely relocated (*i.e.*, will not cause a potential program crash) will be placed in isolated and randomized memory.

**Entities Protected:** All user-space applications that are converted using StackArmor.

**Deployment:** StackArmor requires a binary with debug symbols available for full protection. For stripped binaries, only stack frame randomization and data initialization can be provided.

**Execution Overhead:**

- When tested against the SPEC 2006 CPU Benchmark, StackArmor had an average overhead of 28%.

**Memory Overhead:**

- Worst-case memory overhead on the SPEC benchmark was 195MB, but the authors note that this is highly workload-dependent.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

No modifications are required to the above components. The technique re-writes the binary itself.

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** The implemented version of StackArmor relies on the PEBIL binary instrumentation platform.

**Weaknesses:** StackArmor is designed to increase the granularity of randomization beyond what can be provided by ASLR. To this end, it suffers from the same weaknesses as ASLR: it is vulnerable to memory disclosures and may not actually provide a helpful amount of entropy (this is not analyzed by the authors). In addition, the technique relies on imprecise static binary analysis. This imprecision limits what can actually be randomized without crashing a program on legitimate inputs. Finally, no analysis is provided of how much StackArmor actually impedes ROP-type attacks. For example, a ROP attack inside a single stack frame may not be prevented by any of these techniques.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This increases the granularity of stack randomization. It may provide some barrier to ROP-type attacks, but it is unclear how much added security it actually provides. In addition, attackers can simply use heap-based attacks without being impacted by this defense.

**Availability:** Not publicly available

**Additional Considerations:** None

**Proposed Research:** This technique would benefit from finding a way to efficiently re-randomize memory in order to mitigate memory disclosures. In addition, it would be useful to actually quantify how much randomization is being added above what ASLR provides. If some benefit is observed, extending this defense to the heap may be valuable.

**Funding:** European Research Council

#### 4.1.20 ISOMERON

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** Isomeron [65] defends against control injection attacks, specifically ROP and JIT-ROP [58]. Under such a threat model, it is assumed that the attacker has access to a memory-disclosure vulnerability, by which it may disclose the addresses of gadgets within executable memory. These gadgets can then be leveraged to construct a malicious payload through a script. For example, a website may load malicious javascript, which exploits a memory disclosure to disclose the location of (potentially randomized) gadgets, and then constructs and executes a malicious payload using those gadgets.

**Description:**

**Details:** Isomeron relies on a technique called *program twinning*, in which two copies of the program executable are placed within the address space. One version of the application $A$, is the original program, while the other copy, $A_{div}$, is a diversified copy of the same program. Each of the copies are functionally equivalent, but their gadgets are located in different locations, both absolutely, and relatively. During runtime, execution flips randomly between the two copies. The attacker cannot infer whether $A$ or $A_{div}$ is executing, and therefore cannot predict which version of each gadget should be used. Therefore, the probability of a successful attack diminishes exponentially with the length of the gadget chain.

**Entities Protected:** All programs executed through Isomeron.

**Deployment:** This technique can be implemented on any generic machine. The authors implemented this technique through dynamic binary instrumentation, but note that it could be implemented in the compiler instead. For maximal effect, Isomeron depends upon hardware support (*e.g.*, memory segmentation or Intel SGX) to protect Isomeron-specific data structures, which may not be available on all hardware platforms.

**Execution Overhead:**

- The authors report an average execution time increase of 19% over the PIN dynamic binary instrumentation framework for the SPEC 2006 benchmark.
- Dynamic binary instrumentation itself, as discussed in Section 4.1.12, induces 70% overhead on its own.

**Memory Overhead:**

- The code size of the application is at least doubled to account for the twin program copy. Additional memory is required to maintain Isomeron-specific data structures. The authors did not specifically quantify this overhead.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** This technique, as the authors implemented it, relies on dynamic binary instrumentation, similar to other MTD techniques (*e.g.*, Dynaguard, see Section 4.1.12). It may be possible to implement multiple such techniques on top of a common dynamic binary instrumentation framework, which in some application domains may justify the increased overhead induced by dynamic binary instrumentation.

**Weaknesses:** The primary weakness of Isomeron is its runtime overhead, which renders it untenable in most application domains. Additionally, hardware support, either through Intel SGX or memory segmentation, is required to protect Isomeron-specific data structures. If such data structures are compromised, the attacker could predict or decide which code copy will execute in the future.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** This technique does not fundamentally prevent attacks, however, it makes them rather unlikely to be successful. For example, the authors noted that `CVE-2010-1876` in Internet Explorer could be exploited with a ROP-chain of 6 gadgets. The likelihood of carrying out this attack in the presence of Isomeron is only 1.56%, and attacks that rely on longer ROP-chains are exponentially more unlikely to succeed.

**Availability:** Source code is not publicly available.

**Additional Considerations:** The components of Isomeron must themselves be protected against code and data tampering. This could be accomplished through Intel SGX extensions or segmentation.

**Proposed Research:** The authors indicate that this technique could be implemented in the compiler. This would significantly decrease the overhead of the technique by avoiding the need to

use dynamic binary instrumentation. This is an important avenue for future research, as there are likely independent research challenges associated with such an implementation.

**Funding:** Unknown

### 4.1.21 OPAQUE CONTROL-FLOW INTEGRITY

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** This technique [79] assumes a very strong threat model in which an attacker may find and disassemble all code pages of the victim process. Therefore, the attacker may identify all gadgets. Furthermore, it is assumed that the attacker can read and write the full contents of the heap and stack. This threat model admits code-reuse techniques such as JIT-ROP, as well as attacks against coarse-grained control-flow integrity (CFI) techniques.

**Description:**

**Details:** Opaque CFI (O-CFI) combines coarse-grained CFI with fine-grained code diversity. Coarse-grained CFI checks that the target address of a control-flow transition must be within a valid range (Finer-granularity CFI techniques with higher runtime overhead exist that enable more precise checks of which target addresses are valid.) By randomizing the code-section address space of the program, the bounds of the valid control-flow paths are changed. Each program instantiation is randomized differently, therefore, the set of gadget chains that are considered valid with respect to CFI are different. Therefore, while an attacker may be able to identify all of the gadgets in a program, the program diversity renders the admissible gadget chains different for each program invocation, which thwarts the ability of the attacker to reliably compromise the system.

**Entities Protected:** Any program that is processed by the O-CFI binary rewriting tool.

**Deployment:** This technique could be applied on any generic machine. The implementation the authors present [79], is based on binary rewriting. The authors note that this technique could also be implemented in the compiler.

**Execution Overhead:**

- The authors report an average runtime overhead of 4.7% for the SPEC 2000 benchmarks.
- The authors report an average offline processing time of 5.85s for the SPEC 2000 benchmarks. This processing time is necessary to rewrite a binary to instrument it with O-CFI.

**Memory Overhead:**

- For the SPEC 2000 benchmark, the binary size increased by, on average, 137%.
- For the SPEC 2000 benchmark, the code-segment size increased by, on average, 71%.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(If implemented via binary rewriting, no modifications are required to the above components.)

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** None

**Weaknesses:** This technique is vulnerable to more recent attacks against fine-grained CFI, such as Control Jujitsu [34]. Such techniques exploit the fundamental imprecision of static analysis for the generation of the control-flow graph. Gadgets can be chained together to construct an attack that static analysis cannot prove to be invalid control flow. Such attacks are applicable against O-CFI as well.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** O-CFI adds additional constraints on the attacker by limiting and randomizing the gadget chains that can be legally executed. However, there is evidence to support that legal gadget chains can be constructed to conduct a deterministic attack against O-CFI [34].

**Availability:** Source code is not publicly available.

**Additional Considerations:** At the time of writing, Intel MPX hardware extensions were not widely available. These hardware extensions can be used to accelerate the CFI checks, which would in turn reduce the runtime overhead of O-CFI.

**Proposed Research:** By implementing O-CFI within the compiler instead of via binary rewriting, a more precise control-flow graph is available. As inaccuracy in the static analysis weakens the security of a CFI technique, it stands to reason that the compiler version of O-CFI could offer moderately improved security over the binary-rewriting version presented.

**Funding:** National Science Foundation, Air Force Office of Scientific Research, Raytheon Company, Defense Advanced Research Projects Agency, Mozilla Corporation, and Oracle Corporation

#### 4.1.22 ASLR-GUARD

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** This technique [80] defends against a strong threat model that permits arbitrary memory read/write capabilities. It is assumed that the attacker has access to a copy of the program, on which they can conduct any program analysis. DEP is assumed to be enabled, and therefore the attacker's primary goal is to divert the control flow to execute arbitrary malicious logic.

**Description:**

**Details:** Address-space randomization techniques are fundamentally vulnerable if an attacker can leak the location of attacker-relevant code. ASLR-Guard builds a layer of security on top of address-space randomization techniques in order to prevent the disclosure of *code locators*, such as code pointer or a `vtable` pointer, or other data that can be used to infer a code address. To do so, ASLR-Guard encrypts all code locators before storing them in the data segment of the program's memory space, (*i.e.*, on the stack or heap). In order to accomplish these goals, ASLR-Guard establishes a *safe vault*, where all plain-text code locators are stored. The base address of this region is stored in a dedicated register. Additionally, they designed a technique similar to a shadow stack, which they call AG-Stack, that protect the return addresses and OS-injected code locators. The AG-Stack system maintains two stacks, each of which uses different registers. One stack stores sensitive data such as return addresses and function pointers, while the other stack maintains regular data.

**Entities Protected:** ASLR-Guard protects all programs that are compiled using a compiler with support for this technique.

**Deployment:** This technique could be applied on any generic machine.

**Execution Overhead:**

- The authors report minimal overhead (on average <1%) on the SPEC 2006 benchmarks, though some programs exhibited as much as a 10% increase in execution time.
- The author report 31% increase in the load time on average, or roughly $0.8\mu s$.

**Memory Overhead:**

- The executable file size increase is 6.26% for the SPEC 2006 benchmarks.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique is tightly coupled with the address-space randomization it uses. Incorporating ASLR-Guard with other address-space randomization technique, particularly ones based on re-randomization, may pose significant research and/or engineering challenges.

**Weaknesses:** While ASLR-Guard encrypts all code locators, if an attacker can correlate an encrypted code locator and its associated functionality, it may still be feasible to carry out a code-reuse attack [81].

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique significantly hinders the attacker in conducting control injection attacks against ASLR by making it much more difficult to disclose the addresses of relevant gadgets.

**Availability:** Source code is available at `https://github.com/sslab-gatech/aslr-guard`.

**Additional Considerations:** None

**Proposed Research:** The current prototype only protects static code. A future challenge is to provide the same techniques to dynamically generated code.

**Funding:** National Science Foundation, Office of Naval Research, Department of Homeland Security, United States Air Force, Defense Advanced Research Projects Agency, Korea's Electronics and Telecommunications Research Institute

### 4.1.23 TIMELY ADDRESS SPACE RANDOMIZATION

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Flow Hijacking

**Details:** This technique [82] defends against exploits that hijack control flow by forcing a memory disclosure and using that leaked information to build the main payload. The attacker is assumed to have unlimited access (in both time and space) to arbitrary memory-disclosure vulnerabilities and memory-corruption vulnerabilities anywhere in the program or its libraries and other dependencies. It assumes that the attacker does not have out-of-band access to memory contents from an external source.

**Description:**

**Details:** This technique repeatedly re-randomizes the location of all executable code in a process, including linked code, during runtime. The re-randomization is cued by the operating system based on the tracking of input/output system-call pairs, such that re-randomization is performed immediately before acting on program input whenever a program output has occurred since the most recent re-randomization. Although the threat model assumes that an attacker can trigger a memory disclosure at will, the timing of the re-randomization ensures that any leaked information about the location of program code is obsolete before the attacker can act on it. All executable regions of memory are randomized independently of each other, and the granularity of the randomization is based on the native ASLR randomization of the operating system. The main advantage of the technique is that it synchronizes the defensive action with that of the attacking action rather than performing a one-time randomization step or relying on arbitrary timings and assuming that the attacker will conform to its defensive model. The technique is intrusive, requiring modifications to an application's build process and to the operating system itself.

**Entities Protected:** Programs written in the C language that have been built with an appropriate compiler and run on an appropriately modified Linux operating system according to the specifications of this technique.

**Deployment:** This technique requires a modified compiler and a modified operating system. Modified components are compatible with non-modified ones, meaning that programs built with the modified compiler run normally on unmodified operating systems and unmodified programs run normally on modified operating systems (albeit without extra protections in either case).

**Execution Overhead:**

- Ranges from negligible to approximately 10%, with an average overhead of 2.1%

**Memory Overhead:**

- Ranges from negligible to a few megabytes

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique works in conjunction with the base operating system ASLR implementation.

**Weaknesses:** The principle security weakness of this technique is that it does not re-randomize data locations and is thus vulnerable to attacks that rely on data-only operations. Examples include use of pointers to function pointers, or data-oriented programming techniques [83]. Unfortunately this technique cannot be extended to data location re-randomization while also maintaining compliance with the C standard. The technique is also vulnerable to shortcomings of the base ASLR implementation: in standard commercial implementations of ASLR, partial pointer overwrites — overwrites that target only the lower bits of an address in order to redirect control to another portion of the same segment — are an effective attack. However, by the same token, this technique also gains the benefits of any later ASLR improvement. Performance overhead is modest; however, there are a few legal language constructs that cannot be automatically handled, such as in-line assembly or union type punning. C programs that specifically implement just-in-time compilation techniques cannot be protected at all by this technique.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique prevents attackers from learning any usable information about program code locations using memory-disclosure exploits. Attackers must therefore craft their exploits to be entirely agnostic about the location of program code segments, which has historically required a greater level of effort.

**Availability:** No code is publicly available.

**Additional Considerations:** None

**Proposed Research:** This technique directly benefits from the improvement of base ASLR implementations, which is already an area of research with broad utility. Additionally, a companion technique is needed in order to guard against data-only attacks.

**Funding:** Department of Defense

#### 4.1.24 TIMELY RANDOMIZATION APPLIED TO COMMODITY EXECUTABLES AT RUNTIME

**Defense Category:** Dynamic Software

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection, Control Injection, Data Leakage

**Details:** TRACER [84] mitigates control- and code-injection attacks. The threat model considers an attacker that can interact repeatedly with the target program, and is able to read and write to arbitrary addresses in memory. They have access to both relative and absolute memory operation primitives, and can use both spatial and temporal vulnerabilities. The authors consider both attackers attempting to gain arbitrary code execution, as well as attackers trying to leak sensitive data in memory.

**Description:**

**Details:** This technique re-randomizes the Windows Import Address Table (IAT) of a running process every time a pair of write and read I/O operations occurs. It also re-randomizes the IAT when a configurable time period has elapsed if no I/O has occurred. Attempts to use the original IAT will cause a program crash. Attempts to use the new IATs will be ineffective as re-randomization prevents an attacker from using any function-location information leaked from the IAT in an attack. The protection is accomplished by patching any code in the application that references the original IAT to point it to trampolines that decrypt a protected pointer to the randomized IAT. This patching is done on every re-randomization.

**Entities Protected:** Commodity Windows executables running on the Windows operating system.

**Deployment:** TRACER is a small application installed by an administrator on a Windows computer to be protected. It requires a binary for the application to be protected but does not require source code.

**Execution Overhead:**

- TRACER was evaluated on the SPEC 2006 CINT benchmark. It has an average overhead of around 4% with a maximum of 7.5%.

**Memory Overhead:**

- There is a small fixed memory overhead proportional to the number of modules loaded by a running application under TRACER protection.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique is best paired with other standard Windows mitigations such as DEP and ASLR.

**Weaknesses:** Only protects against attacks that use the IAT directly to look up function locations. Other methods such as walking the Process Environment Block are not considered. Keys used to encrypt trampoline targets are simple to break (XOR).

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** TRACER eliminates the most common vector used by malware for locating Windows function addresses for hooking.

**Availability:** No code publicly available.

**Additional Considerations:** None

**Proposed Research:** Further protections against other function location lookup methods such as PEB should be investigated.

**Funding:** Department of Defense

#### 4.1.25 POLYVERSE

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** Polyverse is a commercial company that has three primary technologies based on moving-target defenses. Since these techniques are closest to Address Space Randomization techniques, we present them collectively here, but some of them also show aspects of dynamic software (*e.g.*, the compiler-based diversification) and dynamic platform (*e.g.*, rapid cycling) properties.

**Description:**

**Details:** Polyverse provides three different products. The first product is a compiler-based randomization technique. This provides an install-time randomization that scrambles the program binary generated from the source code without affecting the semantics of the program. The scrambling can be performed by simply pointing the Linux package manager at the proper repository (a one-line command). The second product applies a similar randomization to closed-source applications where the source code is unavailable (primarily for the Windows operating system). This technique employs binary rewriting to apply a boot-time randomization to the layout and instructions of close-source binaries. The third product is a rapid cycling technology that can be applied to continuously running services (*e.g.*, web servers) to periodically restore their environment to a pristine, good state. It is built using containers (*e.g.*, Docker), and it periodically removes possible attacker persistent foothold on the machine. The recycling is performed through a load balancer that can setup and tear down web-server instances and balance the load among active instances.

**Entities Protected:** All software that has been protected by Polyverse techniques

**Deployment:** Can be deployed on any Windows or Linux machine

**Execution Overhead:**

- Binaries scrambled via the compiler have negligible performance overhead. COTS binaries scrambled via binary rewriting have negligible performance overhead at runtime, but add a five-second latency at startup.

**Memory Overhead:**

- Negligible

**Network Overhead:**

- There is no networking overhead for binary scrambling. Marketing literature reports 13ms additional latency for the rapid cycling technology

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☒ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☒ Persistence

**Interdependencies:** Unknown

**Weaknesses:** Two of the Polyverse products implement one-time randomization. Such techniques are vulnerable to information leakage, in which an attacker may be able to discover the location or content of relevant code to construct an attack. However, unlike traditional ASLR, in which the disclosure of one address gives sufficient information for an attacker to infer the entire program's address space, under Polyverse, an attacker would require far more information to be leaked. For a more complete discussion of the amount of information leakage necessary to conduct an attack under different randomization granularities, we refer the reader to [76].

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** Without information leakage, control hijacking attacks will be significantly hindered by Polyverse techniques as guessing the right scramble is prohibitively hard for an attacker. In practice, attackers have to resort to information leakage to attack the scrambled code. This often requires a separate vulnerability, which raises the attack cost for the attackers, and prevents large-scale compromises.

**Availability:** Commercially available from Polyverse

**Additional Considerations:** All three techniques are built to be easily configurable through a one-line command, which simplifies deployment. There may be modest performance overhead in practice due to load balancing and rapid cycling.

**Proposed Research:** None

**Funding:** Private investors

## 4.2 INSTRUCTION-SET RANDOMIZATION

### 4.2.1 G-FREE

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Instruction Set Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Control Injection

**Details:** This technique [85] aims to mitigate return-oriented programming (ROP) attacks against executables compiled with the modified compiler. It does not fully protect against return-to-libc type attacks where the attacker wishes to execute an entire function from the target program.

**Description:**

**Details:** Return-oriented attacks consist of an attacker redirecting control of a program back into itself at specific useful sequences of instructions. This way, no code needs to be injected but the attacker can still achieve malicious behavior by running pieces of the original executable in the wrong order to achieve arbitrary results. The authors note that all return-oriented programming attacks chain together pieces of code that ultimately each end with a *free branch* instruction. These free branch instructions are specific uses of return or jump instructions [86] where the target of the branch is dependent on a value on the stack or in a register (things that can be compromised by the attacker). If the attacker cannot find any useful code ending in a free branch, then he or she can only execute full function calls like in a return-to-libc attack, effectively eliminating generalized return-oriented programming.

The first step to stopping ROP is eliminating all misaligned free branch instructions. Since modern instruction sets are variable length, an attacker can often take a series of instructions and, by jumping into the middle of one of those instructions, execute an instruction on the CPU that never originally existed in the executable. This new instruction is a combination of the ending bits from one instruction and the starting bits of the next. Any free branch instructions that could be created in this way are a side effect of instruction ordering, and removing them would reduce the number of free branches available to an attacker by a large amount. They must be removed carefully, however, since the program semantics must remain unchanged. The authors accomplish this by scanning for these misaligned free branch instructions and inserting No Operation Performed (NOP) instructions to break them up. NOPs do not effect program execution and so can safely act as buffers to prevent adjoining instructions from incidentally creating a misaligned free branch. Additionally, these NOPs are arranged into a so-called alignment sled, which is a long sequence of NOPs, so that no matter what the alignment was when execution reached the start of the sled, by the time it reaches the end, it will be realigned correctly. This is possible because NOPs are the shortest instruction and eventually execution will align onto one of them and continue normally.

The second protection mechanism used is a careful encryption of the return pointer on the stack. At the function call entry point, the return pointer is encrypted (using XOR with a random key) and pushed onto the stack. A set of instructions is also inserted as a footer, directly above

the return instruction, so that the pointer is decrypted before return is called. If, at any point in the middle of the function, a stack overflow occurs, an attacker could not put a value into the return pointer that would be successfully decrypted into his target address. These two techniques together prevent generalized return-oriented attacks.

**Entities Protected:** Protects all binaries compiled with the modified compiler.

**Deployment:** Can be deployed on any generic machine by modifying the compiler.

**Execution Overhead:**

- Approximately 3% slowdown.

**Memory Overhead:**

- Approximately 26% increase in executable size.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** A ROP protection technique (such as G-Free) should ideally be combined with other memory protection techniques (such as ASLR or function-pointer encryption.)

**Weaknesses:** The encryption used is simply XOR so this technique relies on the fact that the attacker cannot read portions of the memory (memory secrecy) [87–89]. If the attacker could gain access to the return pointer value, he or she could recover the key and forge a new return pointer that would be interpreted correctly by the return instruction.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** Restricts attackers to return-to-libc style attacks where whole functions are used instead of attacks using gadgets or misaligned instructions.

**Availability:** No code publicly available.

**Additional Considerations:** A ROP protection techniques is only effective when it is applied to every application running on a machine. If an application or library is not compiled with this

technique, the entire system is vulnerable to ROP attack. This makes compiler-level defenses against ROP limited in scope. There are similar techniques for protection against specific types of attacks [90] (*e.g.*, spraying attacks).

**Proposed Research:** An operating system-level protection against ROP is necessary to defend against ROP in all the libraries and applications. More importantly, the actual capability of ROP attacks in unknown at this point. More research is required to understand the full power of ROP attacks.

**Funding:** European Union Seventh Framework Programme and European Commission

### 4.2.2 PRACTICAL SOFTWARE DYNAMIC TRANSLATION

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Instruction Set Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection

**Details:** This technique [91] protects against code injection into running binaries from all vectors. It does not protect against return-oriented attacks and assumes that the operating system is secure.

**Description:**

**Details:** Previous Instruction Set Randomization (ISR) techniques have two downsides that make them very unappealing: slow execution due to the requirement for an emulator to run any executable code and a weak encryption function, namely XOR. This scheme fixes the first problem by using a very lightweight virtual machine for execution and the second by switching to Advanced Encryption Standard (AES) for encryption. They use an existing VM called Strata [92] for their ISR scheme, modified to allow for the necessary binary rewriting. When an executable is loaded from disk, Strata encrypts it block by block using AES. During execution, each time the program counter would point to an encrypted instruction, Strata decrypts the block that it is part of and continues by calling the regular fetch instruction. Each instruction also comes with a tag that can be verified so that after decryption, Strata can decide whether the code is legitimate or if it has been injected. Any injected code could not match the tag, let along produce a valid, useful instruction for the attacker since he does not know the encryption key used. To speed up execution, once the blocks are decrypted they are kept in a cache for reuse. The encryption key is generated fresh for every program execution and is kept by the VM so it cannot be read or altered by the program.

**Entities Protected:** All executables running on the Strata virtual machine.

**Deployment:** Can be deployed on any generic machine by adding an extra virtualization layer.

**Execution Overhead:**

- Up to a 20% slowdown in execution.

**Memory Overhead:**

- Up to a 70% increase in executable size overhead and memory footprint.

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** Because of relying on a virtualization layer, this technique is really a stand-alone technique that does not combine well with other OS-level defenses.

**Weaknesses:** AES is used in Electronic Codebook (ECB) mode that encrypts two identical blocks to the same value. This means that an attacker could execute a replay attack by finding useful encrypted instructions that exist in the executable and injecting them as shellcode. ECB is used for efficiency reasons so that fewer decryptions are required. Additionally, the Strata VM itself becomes a new point of attack since it holds all the keys and is in charge of readying instructions for execution.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☐ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** Makes it difficult for an attacker to inject arbitrary code into an executable. He or she would need to brute force the encryption key in order to forge instructions that would decrypt to anything useful. More likely, the attack vector will shift to return-oriented programming which is not mitigated with this technique.

**Availability:** No code publicly available.

**Additional Considerations:** The memory and execution overhead may become significant if all applications are virtualized in this manner. A similar technique is proposed in [93, 94].

**Proposed Research:** As with other techniques using encryption, this could benefit from hardware AES instructions recently added to Intel processors.

**Funding:** Defense Advanced Research Projects Agency, National Science Foundation

### 4.2.3 RANDSYS

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Address Space Randomization and Instruction Set Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [95] defends against code injection and control injection from buffer overflow attacks on the stack and heap. This method is only focused on remote machine-code injection attacks. This method also assumes that the kernel is safe and it would not protect against kernel-level code injection attacks.

**Description:**

**Details:** This is a hybrid Instruction Set Randomization (ISR) and Address Space Layout Randomization (ASLR) technique. It uses subsets of techniques from each category along with some additional guards to create a new implementation.

For ISR, it implements system call randomization between user space and kernel space (similar to [96]). When a process is created, the exec system call is intercepted in the kernel and control is given to RandSys. RandSys searches for all system calls in the application then takes their location in memory and generates a new, random system call number using a secret key stored in kernel space. This requires rewriting the system call dispatcher in the kernel to decrypt the system call numbers at run-time.

For ASLR, it implements library re-mapping and function randomization. Library re-mapping randomizes the library base addresses and reorganizes the internal functions. This makes it more difficult to predict both the absolute and relative addresses. The import and export function tables used by the dynamic linker are also randomized. The function randomization makes the name lookup of each function unique to each process. Different randomization algorithms are used depending on whether the function is being imported or exported. Due to this, a separate function name resolver needs to be created to tie the imported and exported function names back together at run-time.

Additional protections are also implemented with RandSys. Decoy entries are placed in the function import and export tables. Each decoy points to a guard page which will cause an access violation exception if there is an attempt to read, write, or execute it. RandSys also implements a method for dynamic injection detection. A code page with injected shell code will have two properties that can be detected: it will be writable and it will not be mapped from the executable file. Whenever a system call or library function is invoked, a recursive stack-based inspection algorithm can determine if any of those code pages exist. It hooks into the exception handler and watches for such exceptions. It will attempt to terminate any program that has such an exception.

**Entities Protected:** All programs running on a machine utilizing this technique.

**Deployment:** Can be deployed on any generic machine by modifying its operating system.

**Execution Overhead:**

- Increased system call overhead (difficult to estimate but could increase execution overhead by up to 20%).
- Additional overhead introduced by one-time disassembly/analysis of each executable, up to several minutes per executable.

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** The ASLR implemented by RandSys cannot be combined with another ASLR implementation, so one has to be selected. Also it is desired to combine a solution like RandSys with a ROP protection technique.

**Weaknesses:** This defense can be circumvented with a return-oriented programming attack that can find the location of the randomized libraries (through an independent leakage attack, other violation of memory secrecy or brute force).

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** Makes it difficult for an attacker to inject code into a running program and increases the level of effort required to redirect program control to a chosen location.

**Availability:** This implementation has been prototyped for both Windows and Linux but there is not a publicly available version of it.

**Additional Considerations:** This technique breaks self-modifying codes. It also requires an additional disassembly step for each application.

**Proposed Research:** RandSys mainly protects system calls. An extension to RandSys that protects other library calls is an open problem (See [97]). In addition, this type of protection does

not prevent ROP attacks. A complete protection against typical code injection and ROP attacks is an open problem.

**Funding:** National Science Foundation, Microsoft Research

### 4.2.4 RANDOMIZED INSTRUCTION SET EMULATION (RISE)

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Instruction Set Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection

**Details:** This method [98, 99] is targeted at stopping external binary code injection into an executing program. The keys used for randomization are stored in the same memory space as the running process so it relies on the assumption that the process memory cannot be read by an attacker.

**Description:**

**Details:** RISE is a software-based ISR technique built on top of the open source Valgrind IA32-to-IA32 binary translator. It scrambles the instruction set at load-time and descrambles them at run-time. It runs in user-space and does not require any modification of the operating system or program being run because it is running inside an emulator. It can be run on a per-program basis so it does not interfere with programs like compilers. RISE scrambles all executable portions of a process, including libraries, by XOR-ing each byte of the process' code with a randomization mask. RISE has two methods of randomization. The first method is a tiled method that involves generating a random mask with two or more pages before execution and XOR-ing each byte in the code with a byte in the mask. The mask is read from `/dev/urandom` and is stored in a fixed location right before the executable. The second method uses a one-time pad by using a unique mask for each code page. The masks are not generated until the page is first accessed.

In both cases, any code that is injected into the program will be decrypted using the masks and likely result in an invalid execution. For an attacker to circumvent this, he or she would have to be able to generate a code segment that decrypts correctly into another one with his or her desired behavior. Ideally, this can only be done if the attacker discovers the encryption keys.

**Entities Protected:** Any program running inside the RISE emulator.

**Deployment:** This technique can be deployed on any generic machine by adding an emulator.

**Execution Overhead:**

- Additional 5% increase in overhead on top of Valgrind overhead
- Valgrind adds a minimum of 400% overhead per the documentation

**Memory Overhead:**

- Each process creates a private copy of all loaded libraries in virtual memory
- The One Time Pad randomization doubles the amount of memory needed for the code

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification required)

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** RISE relies on a emulation layer (Valgrind). If this is to be used for all of the applications, the overhead will be significant.

**Weaknesses:** This framework does not protect against attacks that target functions or pointers, including return-oriented programming attacks. Additionally, an attacker that can violate memory secrecy could read the key directly from memory or recover an encrypted code segment that, along with the unencrypted segment obtained from the original executable, can be used to deduce the key.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique makes it difficult for an attacker to inject viable shellcode into an application running inside the RISE emulator, without having an independent vulnerability that can violate memory secrecy.

**Availability:** Prototype available under GPL at `http://cs.unm.edu/~immsec`

**Additional Considerations:** The large overhead introduced by the emulation layer can make RISE impractical for real-world applications. See [100] for a discussion of performance issues.

**Proposed Research:** Similar to the function pointer encryption technique above, RISE could benefit from the hardware level AES instruction providing an encryption scheme resistant to the known plaintext attack outlined above.

**Funding:** National Science Foundation, Office of Naval Research, DARPA, Sandia National Laboratories, Hewlett-Packard, Microsoft Research, Intel Corporation

### 4.2.5 SQLRAND

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Instruction Set Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection

**Details:** SQLRand [101] aims to protect against Structured Query Language (SQL) injection attacks in situations where the query depends partially on untrusted input.

**Description:**

**Details:** SQLRand is a system for randomizing the SQL query language to prevent SQL injection attacks. The creators note that injection attacks on SQL can be thought of similarly to buffer-overflow-based code injection attacks. Their methods for SQL are based on similar methods in RISE for such attacks. The SQL language is randomized so that any code that was injected will not run (it will not match the new randomized language). A base SQL query (without runtime criteria derived from user input) is sent to a proxy server to be randomized and returned. The randomization is done by appending a chosen integer to the end of every keyword in the SQL language. When the query is executed, it is again sent to the proxy that derandomizes it and passes it on to the database server. Any code that was injected into the query by the user will not match the new randomized language and will cause the query to fail.

**Entities Protected:** Any database application that uses the SQLRand proxy.

**Deployment:** Can be deployed on a network as a standalone proxy or on the machine that runs the database software. Requiring use of the proxy to access the database would increase security.

**Execution Overhead:**

- The randomization is relatively simple and very fast; experimental query response times were increased by 6 milliseconds.

**Memory Overhead:**

- None

**Network Overhead:**

- Requires a proxy for all traffic going to the database server

**Hardware Cost:**

- Can be run on the same server as the database software, but could also be run as an independent server for increased speed

**Modification Costs:**

- ☐ Data
- ☐ Source Code
- ☐ Compiler/Linker
- ☐ Operating System
- ☐ Hardware
- ☐ Infrastructure

(No modification required)

**Expertise Required to Implement:**

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☒ Custom Programmer (General Knowledge)
- ☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

- ☐ Seamless
- ☒ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** None

**Weaknesses:** If the randomized SQL query is ever leaked or accessed by the attacker then he or she can produce valid injection code. This is very common with web applications that often report the query used upon failure. Developers would have to be very sure that error messages were sanitized and no other paths for query leakage were introduced. However, since most SQL injection attacks start by discovering a query (otherwise the attacker would have no knowledge of the database structure), this seems like a very large weakness.

**Types of Weaknesses:**

- ☐ Overcome Movement
- ☒ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** Increases the level of effort for SQL injection attacks by making it more difficult for attackers to generate valid injection code. It requires them to either brute force the random key or find a way to leak an existing randomized query.

**Availability:** No code is publicly available.

**Additional Considerations:** This approach requires every use of a SQL query to be rewritten (in the source code) with this randomization in mind. In particular, the developer must identify the parts of the query that will always remain the same and the parts that are based on user input. Since the vast majority of SQL injection attacks occur because the developer did not take the time to do this in the first place (if he or she did there is already a method for sanitizing inputs using the prepare command), this seems like a wasted effort. Moreover, the scope of the protection is also very limited.

**Proposed Research:** There are existing, effective techniques to stop SQL injection attacks. No research is proposed.

**Funding:** Unknown

### 4.2.6 CIAS

**Defense Category:** Dynamic Runtime Environment

**Defense Subcategory:** Instruction Set Randomization

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection

**Details:** This technique [102] protects against the injection of code into an application with a buffer-overflow vulnerability. This technique is only effective against injected code that requires the use of system calls.

**Description:**

**Details:** First, the compiler is modified so that each system-call number is changed from $x$ to $f(r, x)$ where $f$ is a random permutation that takes $r$ as an input seed. In practice, they use XOR as $f$. This means that every system call number is replaced by a randomly chosen pseudonym. Any code that is injected will not know this mapping and thus cannot produce shellcode that invokes the correct system call. The kernel system call dispatch is changed so that it knows $f$ and $r$ and can derandomize the input number to the correct system call number.

**Entities Protected:** Any programs recompiled using the modified compiler, on a system that includes the kernel derandomizer.

**Deployment:** Can be deployed on any generic machine by modifying the operating system.

**Execution Overhead:**

- The kernel must derandomize system call numbers, but system call dispatch already takes a significant amount of time and one additional XOR does not have significant impact.

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** Could be combined with ASLR to increase protection.

**Weaknesses:** The system-call table is not very large so the amount of randomness introduced is small (can be as low as 8 bits). Additionally, if a randomized binary is leaked then an attacker can compare that to a regular binary and discover the key, gaining the ability to forge system call

numbers. Also, this does not protect against return-oriented attacks because the system calls in libc will already be correctly randomized.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** Increases the effort required to inject the desired code successfully.

**Availability:** No code publicly available, but the concept is relatively simple and would not require many code changes.

**Additional Considerations:** None

**Proposed Research:** As with other techniques that use XOR as an encryption function, this could possibly benefit from hardware AES. This may be a better research opportunity because system calls happen relatively infrequently (compared to pointer dereferences) and already require a shift to kernel space. This means that any performance degradation will be well hidden and the problem of storing keys is dealt with because they can be securely stored in kernel space. However, this solution is a partial solution to a bigger problem. A proper memory protection against regular code injection and ROP is required.

**Funding:** National Natural Science Foundation (China), Beijing Science Foundation, Nation 868 High-tech Program of China, MOE Key Laboratory of Data Engineering and Knowledge Engineering

# 5. DYNAMIC PLATFORMS

## 5.1 SECURITY AGILITY TOOLKIT

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Exploitation of Trust

**Details:** This technique [103] helps mitigate the damage that can be done on a system by restricting the access an application or process currently holds in the event of attack detection. It can restrict high-level access like read/write permissions to a file as well as low-level access such as system calls. It also has the ability to restrict external connections.

**Description:**

**Details:** This technique provides a toolkit to wrap around executables. It allows the injection of greater access control mechanisms with the ability to change them during program runtime. The toolkit is meant to supplement general intrusion detection system (IDS) frameworks. The idea is that if a detection of a certain threat or activity is encountered, the dynamic security policy of the affected applications can be dynamically changed. It could increase auditing, isolate affected processes, or even take measures like killing certain programs. There is an Agility Authority on each host that manages the agile processes for that host. Above that, an Agility Authority Manager distributes policy updates to each Agility Authority. The IDS can either send response directives directly to each Agility Authority or to the Agility Authority Manager. After a response directive is received, the policy is adjusted accordingly and actions are taken according to those new policies to mitigate the threat.

**Entities Protected:** This technique protects the operating system when suspicious activity or threats are detected.

**Deployment:** This technique could be implemented into the operating system at the kernel level to enable functionality to wrap around existing executables.

**Execution Overhead:**

- Will incur some unknown overhead while checking for policy updates and applying policy checks

**Memory Overhead:**

- Will incur some unknown overhead by injecting the policy code into the running program

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☒ Persistence

**Interdependencies:** This technique relies on a framework that includes intrusion detection systems, event analyzers, and response units to trigger policy changes. If the attack cannot be detected, this framework does not work.

**Weaknesses:** A potential weakness is the reliance on a separate detection mechanism. A stealthy attacker could avoid detection and carry out their attack without extra hindrance. An attacker could also potentially use the policies to cause a denial of service to the system by intentionally triggering the strict policies. This technique does not provide any protection against the first attack. It can only adjust the policy afterward.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☒ Disable Movement

**Impact on Attackers:** Depending on how this was implemented, it may have some impact on the attackers. If the policies were implemented in a strict fashion as the starting policy, it could limit what the attacker could do to a system after compromising an application. If the attacker is detected, it could make it more difficult accomplish their task if they had not completed it before being detected.

**Availability:** This technique was prototyped by the authors but is not publicly released.

**Additional Considerations:** This work lacks many specifics. The technique relies on an external detection mechanism so the effectiveness of this technique relies on the effectiveness of that mechanism. Since there is no perfect detection mechanism, this could significantly decrease the effectiveness of this technique. Also the technique does not provide any protection against the first attack. It only relies on limiting damage afterwards. Moreover, policy changes can break applications and functionality.

**Proposed Research:** A possible future direction for this technique would be to make it more integrated with a detection mechanism. This would make the technique less reliant on external triggering mechanisms. Combining this technique with other movement techniques would increase the overall effectiveness of this method. If other techniques or guards are able to detect more specific types of attacks, that could be implemented as another triggering source for this technique. It is also crucial to understand the impact of policy changes and their effectiveness in stopping an attack.

**Funding:** DARPA

### 5.2 GENESIS

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [104] defends against different threats depending on how it is implemented. If it is implemented with Instruction Set Randomization, it can defend against code-injection attacks. If it is implemented with Calling Sequence Diversity, it can partially mitigate attacks that divert the control flow of a program. It can protect against attacks that target function calls that exist before the program is loaded but not function calls that are generated during runtime.

**Description:**

**Details:** This technique involves applying runtime software transformations to a program. The program is run in an application-level virtual machine called Strata. Strata with software dynamic translation can change a program by injecting new code, modifying existing code, or controlling program execution in some manner. Strata examines and translates all program instructions before they execute on the host processor. Two transformation methods were prototyped to test this technique.

The first method involves Calling Sequence Diversity (CSD). The method involves modifying the compiler to insert annotations into the code whenever there is a static control flow switch. It will XOR three keys each time this switch is made and will be compared to an expected key to verify it was a valid switch. The first key is generated at load time and is not accessible by or stored in the running program. The second and third keys are the source and destination keys. If an unexpected jump or control flow diversion is interested, Strata will dynamically generate the key check.

The second method involves modifying the linker to allow Strata to use Instruction Set Randomization (ISR). This method uses 128-bit AES encryption instead of XOR. The linker marks all application and library code as encryptable, appends a tag to each instruction, and adds padding as necessary to properly align the blocks for AES encryption. Strata will encrypt the application when it loads and decrypts the instructions as they are needed for execution. It will then check the instructions for the proper tag. If the instruction is valid, it will remove the tag and add it to a cache to decrease decryption costs.

**Entities Protected:** This technique helps protect the operating system by making applications more difficult to exploit.

**Deployment:** The Strata VM is deployed as a standalone application on the system and does not require modifying the operating system. In order to use the methods of diversity discussed in the paper, the compiler and linker on the system would also need to be modified to support each diversification method.

**Execution Overhead:**

- The ISR method adds about a 17% increase in overhead against the SPEC CPU2000 Benchmarks
- The CSD method adds an average 54% increase in overhead against the SPEC CPU2000 Benchmarks
- The emulation layer (Strata) can impose significant execution overhead

**Memory Overhead:**

- Some additional memory will be required for the Strata VM and any keys or instructions it needs to store

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

- ☒ Seamless
- ☐ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☐ Attack Launch
- ☐ Persistence

**Interdependencies:** This technique relies on an emulation layer (Strata).

**Weaknesses:** This technique relies on the security and integrity of the VM. It assumes there is no way for an attacker to disable protections on the VM's memory sections and the VM implementation is sufficiently bug-free. The authors claim to protect the system calls that could disable these protections but a method may exist to disable those protections indirectly. In addition, this technique does not provide any protection against ROP attacks. This technique is also weak against memory secrecy violations.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☐ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** Both methods used in this technique will increase the amount of work needed to exploit the application. In both cases, more advanced control injection attacks, such as return-oriented programming, could be used to bypass these protections.

**Availability:** This technique was prototyped by the authors but is not publicly released.

**Additional Considerations:** The ISR method breaks self-modifying code and just-in-time compilers (*e.g.*, Java). Running every application on top of an emulation layer can have significant execution overhead, which makes this technique impractical.

**Proposed Research:** A future research direction for this technique might be investigating methods to increase the protection provided by the CSD method. A larger direction might be combining this technique with an N-version programming technique. This would increase the overall difficulty in exploiting the application because now the attacker has to break multiple diversifications with one input. An efficient protection against code injection and ROP is an open problem.

**Funding:** DARPA

## 5.3 MULTIVARIANT EXECUTION

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection

**Details:** This technique [105] combats code-injection attacks by having each running variant use a different system call mapping and unpredictable stack direction. Each variant uses the same input making it difficult to inject code that will work with all mappings simultaneously. The stack direction change will result in a different flow of instructions in the program and libraries as well as providing different library entry points between the variants.

**Description:**

**Details:** This technique involves running multiple variations of the same program. A separate monitoring program monitors all variations. The level of monitoring can vary from each program having the same result down to checking each instruction executed. This technique focuses on synchronizing all variants at the system call level and each variant should make the exact same system calls. If any inconsistency is detected, all variants are terminated and restarted. The monitor is implemented as a user-space, unprivileged program. It monitors the arguments of system calls and all communication between the variants as well as interactions with the kernel. There are some system calls that must be executed by the monitor on behalf of the variants to keep them synchronized. These would include system calls that change the state of the system or return volatile results. In this case, the results of the system call are passed to the variants. Variants are automatically generated by modifying the stack growth direction and system call number mapping but the technique is capable of any variation technique as long as the system call invocations are the same.

**Entities Protected:** This technique protects the operating system by making the exploitation of an application more difficult.

**Deployment:** The monitor and variants are implemented as standalone applications running on a system. The variants are generated by using a modified compiler and modified system libraries.

**Execution Overhead:**

- Number of variants + Monitor overhead
- Additional time added for variant synchronization and communications
- Average monitor overhead of 17% with two variants
- Average monitor overhead of 30% with three variants
- Average monitor overhead of 37% with four variants

**Memory Overhead:**

- Number of variants + Monitor overhead

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

**Interdependencies:** This technique requires a good source of randomness for the system call number randomization. This technique will not work with variants that cause a program to produce differing sequences of system calls.

**Weaknesses:** The actual dependency of attacks on the variants is unknown. This technique does not stop attacks against higher-level protocols. The multivariant monitor can be compromised specifically as it is in the "untrusted" zone. For example, the monitor can falsely indicate that the variants agree on a result. In addition, this technique is focused on integrity attacks and it does not protect against data leakage (exfiltration) attacks against one of the variants. The granularity of detection is also limited to system calls. Modifications to the user space code that keeps system calls intact remain undetected.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** This technique increases the difficulty of exploiting an application. It requires the attacker to provide input to a program that will simultaneously break all running variants. The impact will be different depending on the diversification method used on the variants. Some diversification techniques include stack reversal, instruction set randomization, heap layout randomization, stack base randomization, variable reordering, system call number randomization, register randomization, library entry point randomization, stack frame padding, code sequence randomization, equivalent instructions, program base address randomization, program section reordering and program function reordering.

**Availability:** This technique was prototyped by the authors but is not publicly released.

**Additional Considerations:** The technique has significant overhead as it requires many variants. It, however, does not protect against the compromise of one variant. The actual impact of diversity

on the attacks is unknown. This technique also lacks many important specifics (types of diversity applied and its impact).

**Proposed Research:** It may be possible to compose some techniques while still preserving the required properties of this technique. It would be worthwhile to explore which methods can be composed together in a manner that does not cause unintentional divergences or false detections. More specifics are needed for a technique like this.

**Funding:** Air Force Research Laboratory

## 5.4 DIVERSITY THROUGH MACHINE DESCRIPTIONS

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection

**Details:** This technique [106] is meant to mitigate mass code injection attacks. Each system would potentially need their own custom exploit to work because of all the varying system modifications and configurations.

**Description:**

**Details:** This technique involves using a virtual machine and compiler machine descriptions to create a diverse set of architectures. This will regenerate all the machine-dependent and architecture dependent parts of a complete operating system. Various items can be changed in these machine descriptions including the following:

- different size of operations with different instruction sets and instruction encoding
- different number of registers
- different machine byte and word sizes
- different endiannesses
- different representation of signed integers
- different stack directions
- using one or multiple stacks
- different calling conventions such as alignment, ordering, padding, registerization, stack adjustment, and return value handing
- alignment padding in stack frames and data structures

The kernel would be able to have changes such as different sizes of standard types and linker relocation codes. These machine descriptions could be randomly generated. These architectures could be periodically applied to one machine or across many machines.

**Entities Protected:** This technique protects the operating system as a whole.

**Deployment:** This technique is deployed inside of a VM and is composed of an entire system.

**Execution Overhead:**

- Some overhead imposed by running inside a virtual machine

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique relies on a virtualization layer. It also require a diversification layer to create the variants.

**Weaknesses:** The technique does not protect against application-level attacks. It does not protect against ROP attacks either. The virtualization layer is also a single point of failure and can be attacked. In addition, the technique does not provide any protection against targeted attacks on one platform.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☒ Disable Movement

**Impact on Attackers:** This technique would increase the amount of work an attacker would have to do to attack a large number of systems. An attacker may still be able to leverage non-code injection attacks against these systems that work across multiple architectures.

**Availability:** This is a research idea proposed by the author and has not been implemented.

**Additional Considerations:** This technique is theoretical and lacks many specifics. Constant recompilation of the entire operating system could impose a large operational overhead In addition, changing so many aspects of a system could potentially have unforeseen adverse effects on applications and can break functionality. Implementing this technique can be very difficult. See a discussion on diversity in [107, 108].

**Proposed Research:** A possible research direction for this technique might be coming up with a way to better automate this process. It may also be worthwhile to investigate potential side effects of changing so many parts of the architecture. This idea might be able to be expanded to mix in different versions of libraries and base operating systems as well.

If a technique like this could be reasonably automated (see [109]) and any side effects of architecture randomization explored, a larger future direction could be combining this with a technique like TALENT. This would allow for a large space of platforms to be dynamically generated or periodically regenerated.

The impact of different types of diversity on attacks has to be studied.

**Funding:** Unknown

### 5.5 N-VARIANT SYSTEMS

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [110] can be implemented with different application variants to target specific threats. The instruction set tagging variant gives each running variant their own instruction set. Since each variant is passed the same input, this will help mitigate code injection attacks because the attack might succeed on one variant but would presumably fail on another. The monitor would catch this divergence and restart the variants. Address-space partitioning is the second type of variant used in this technique. This variant will help mitigate control-injection attacks because each variant is mapped to a separate location in memory. This will only help mitigate control flow attacks that rely on fixed addresses. Attacks that know the relative location of their target can still succeed.

**Description:**

**Details:** The idea behind this technique is to run multiple variants of the same application simultaneously without relying on anything to be secret. It contains a polygrapher, the application variants, and the monitor. The polygrapher takes input and passes it to all the variants. The monitor watches the variants for a divergence and, if one occurs, restarts all the variants in a known good state.

This technique relies on a couple properties to work correctly. The first property is a normal equivalence that says when a variant is in a normal state, that state should be equivalent to a state in the unmodified, original process. The second property is a detection property that says certain attacks should be detected as long as the normal equivalence is satisfied. If a variant enters a compromised state, then another variant should enter an alarm state or anomalous state that is detectable by the monitor.

The proof-of-concept was built into the Linux kernel and tested with two types of variants. The monitor synchronizes the variants at the system-call level. System-call wrappers are created so system calls can be shared between variants. System calls are broken into three categories: shared, reflective, and dangerous. Shared system calls interact with external state, reflective system calls observe or modify properties of a process, and dangerous system calls can break the assumptions of the technique.

The first variant tested was the address space partitioning. This utilizes the linker to load the data and code sections of the program at sufficiently different addresses while ensuring they will not overlap. The second variant tested was instruction set tagging. This utilizes a binary rewriter to insert a tag into each instruction and software dynamic translators to interpret these instructions during execution. Each variant would use different tags.

**Entities Protected:** This technique protects the operating system by making the exploitation of an application more difficult.

**Deployment:** This technique is built into the operating system and wrappers are created for some system calls.

**Execution Overhead:**

- $N$ times slow down for $N$ variants plus additional overheads as follows:
  - 2 Variants with Address Partitioning: Unsaturated Server: 17.6% Increase, Saturated Server: 48% Increase
  - 2 Variants with Instruction Tagging: Unsaturated Server: 28% Increase, Saturated Server: 37% Increase
- CPU-bound services will have a high overhead because each variant will duplicate computations

**Memory Overhead:**

- Number of variants + Monitor + Polygrapher Overhead

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** This technique can only be used with diversification techniques that use variants similar enough to satisfy the normal equivalence property. It also relies on separate diversification techniques.

**Weaknesses:** This technique does not protect against application level attacks. Also the monitor can be a single point of failure. In addition, the technique does not provide any protection against ROP attacks and memory secrecy violations. It does not provide any protection against data leakage attacks on one variant either.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique increases the difficulty of exploiting an application. It requires the attacker to provide input to a program that will simultaneously break all running variants. The impact will be different depending on the diversification method used on the variants.

**Availability:** This technique has been implemented by the authors and is available at `http://www.nvariant.org`

**Additional Considerations:** The technique lacks many specifics and the actual diversification techniques and their impacts are unknown. Also the technique kills programs that use the `execve` (execute program) or unrestricted mmap (map file into memory) system calls. Operating System Signals may cause the variants to diverge because variants might be in slightly different states when they receive the signal (this restricts the functionality of the technique). Variants using user-level threading may cause false detections because of the difference in thread interleaving causing different sequences of system calls. This could also potentially allow an attacker to exploit race conditions. Various non-attack inputs can cause false detections making this prototype less feasible for real services. In addition, running many variants may be impractical.

**Proposed Research:** A future direction for this technique would be to explore additional variant-diversification methods. This technique could also be enhanced to work with more system calls and operating system components like other similar techniques. The impact of diversification techniques on attacks must also be studied.

On a larger scale, it may also be possible to compose some techniques while still preserving the required properties of this technique. It would be worthwhile to explore the space of diversification methods and determining which methods can be composed together in a manner that does not cause unintentional divergences or false detections. Some diversification techniques include stack reversal, instruction set randomization, heap layout randomization, stack base randomization, variable reordering, system call number randomization, register randomization, library entry point randomization, stack frame padding, code sequence randomization, equivalent instructions, program base-address randomization, program-section reordering, and program-function reordering.

**Funding:** DARPA, National Science Foundation

### 5.6 TALENT

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection, Control Injection, Scanning, and Supply Chain

**Details:** This technique [111] can help mitigate OS- and architecture-dependent attacks. Since the application is migrating between systems with different libraries, architectures, and layouts, it is more difficult to construct exploits that will work under every platform. The attacker also may not be able to predict when the application will migrate or which platform the application is currently running on. The fact that the application can be constantly moving makes passively scanning and collecting information less useful. Changing hardware platforms also makes supply chain attacks more difficult.

**Description:**

**Details:** The Trusted dynAmic Logical hEterogeNeity sysTem (TALENT) is a technique that involves making a running application migrate between different platforms (OS and CPU architecture) while preserving the state of that application. This state can include any files the program was using or sockets the program had open. These platforms have hosts with virtual containers. Each can be implemented with a different operating system, different hardware, a different architecture, and different versions of libraries. The application being preserved is precompiled for each platform. TALENT needs compiler support to create checkpoints and containers to preserve the environment.

The current implementation uses Linux and BSD platforms. A centralized controller manages the migrations. The migrations can currently trigger at random intervals or via manual interaction.

**Entities Protected:** This technique protects the operating system and applications running on it by continually shifting the attack surface

**Deployment:** This technique is deployed across multiple systems. A special compiler allows a program to be periodically checkpointed. The source code of the program needs to be modified to be able to support the checkpointing.

**Execution Overhead:**

- Minimal overhead imposed by the checkpointing mechanism
- A few seconds of downtime during migration

**Memory Overhead:**

- None

**Network Overhead:**

- The state of the programs will be transferred between machines
- Control messages passed between the platforms

**Hardware Cost:**

- A system capable of a virtual infrastructure or additional machines to host each platform

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** This technique relies on a detection mechanism for effective jumping if it is not using a random jumping scheme.

**Weaknesses:** There are a couple ways this technique could be less effective. If the platforms do not migrate fast enough, the attacker may be able to get an exploit together and attack the current machine. The technique does not provide any protection against higher-level protocol attacks. It also does not protect against attacks on one machine.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☒ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique would slow down an attack and make it more difficult for them to exploit the machines but it would not stop them completely. An attacker still has the chance of exploiting a system and achieving their goal before the application migrates. An attacker could also try to leverage more advanced Control Injection attacks that could work across numerous systems.

**Availability:** This technique has been prototyped by the authors and is available as a GOTS product.

**Additional Considerations:** Using this technique on every application can impose a very high overhead. It must be used to protect only selected set of important applications.

**Proposed Research:** There are a number of future directions this technique could investigate. One direction would be implementing a recovery mechanism. This would allow the technique to clean up and recover from attacks. Another future direction would be adding data integrity checks. Currently the technique has no integrity guarantees. Finally, another direction would be creating a distributed command and control mechanism to eliminate the single point of failure.

A possible direction in the future may be able to combine this technique with a cloud concept to have a large and dynamic set of platforms to choose from at all times. This would make it less predictable which platforms would be in the migration sequence.

The impact of OS and architecture diversity on attacks must also be studied.

**Funding:** Air Force

## 5.7 INTRUSION TOLERANCE FOR MISSION-CRITICAL SERVICES

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource

**Details:** This technique [112] combats resource attacks such as denial of service (DOS) and data integrity attacks. It mitigates the impact of DOS attacks by trying to ensure there are enough resources on a platform to run the service. It will terminate non-critical services to free up resources. If not enough resources can be freed, it will change to a different platform. This technique mitigates data-integrity attacks by implementing a voting scheme for the service results.

**Description:**

**Details:** This technique aims to make critical web services more survivable in the face of attack. This is composed of a frontend that accepts requests from clients, some number of diverse platforms serving the same service, and a surveillance node that monitors the platforms and deals with voting. The platforms can have different operating systems and different web servers to vary their attack surface.

Each platform implements a resource reallocation method that monitors the system. It is composed of a resource-reallocation manager, health-monitor thread and survivability-evaluation thread. The health monitor collects performance information from the operating system and forwards that information to the survivability thread. This thread determines if resources need to be changed based on the performance. If resources need to be adjusted, the resource manager will start to terminate non-critical services to free up additional resources. If not enough resources can be freed to ensure acceptable performance, the platform is taken offline for recovery.

Each platform also has a result acceptance tester. This component tests the logical reasonableness of the result and the execution time required to obtain that result. The platforms can operate in two modes. There is an active mode where a set of active nodes process a request simultaneously and the result is voted on and processed by the surveillance node. There is also a passive mode where only one platform is active. If that platform does not pass the acceptance test, it is replaced by another platform and recovery is performed on it.

**Entities Protected:** This technique protects specific applications and services to ensure they continue to operate under attack.

**Deployment:** This technique would be deployed in the overall network infrastructure.

**Execution Overhead:**

- Up to an additional 50% time may be needed to process a request

**Memory Overhead:**

- None

**Network Overhead:**

- Additional out-of-band network needed for surveillance

**Hardware Cost:**

- Additional platforms to host the additional variants
- Additional network infrastructure to support this platform configuration

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☒ Persistence

**Interdependencies:** The system relies on a detection capability that monitors the resources.

**Weaknesses:** This technique assumes that only one active platform will be compromised at any given execution cycle. The voting mechanism would see this as a valid result and it could compromise the integrity of this technique. Also this technique does not stop any attack. It just tries to mitigate DoS attacks by resource management. In addition, data-leakage attacks or low-observable attacks are not mitigated.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☒ Limit Movement

☒ Disable Movement

**Impact on Attackers:** This technique would make it more difficult for an attacker that is trying to deny service. This technique might not slow down an attacker with a different goal. An attacker could still carry out other attacks that take advantage of the voting system or exploit the system at a lower level.

**Availability:** This technique was prototyped by the authors but is not publicly available.

**Additional Considerations:** This technique provides no additional protection. It just mitigates the impact of certain types of attacks. Also the technique lacks specifics. For example, it is unclear what types of resources are monitored. What happens to obscure resources that can run out (*e.g.*, file descriptors or certain ID numbers)? A similar technique is proposed in [113].

**Proposed Research:** A possible enhancement to this technique would be to make the voting system more difficult to bypass. Currently, only two results need to match for a result to be accepted. This could be expanded to more systems.

A larger direction for this technique would be to combine this technique with other movement techniques on the platforms. This would increase the diversity between each platform and make the platforms resistant to a larger set of attacks.

Also a study must be conducted to enumerate all possible resources that can be attacked in a machine during a DoS attack.

**Funding:** University IT Research Center Project, University of Incheon, Korea Information Security Agency Research Project

## 5.8 GENERIC INTRUSION-TOLERANT ARCHITECTURES FOR WEB SERVERS

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection, Control Injection, and Scanning

**Details:** This technique [114] helps reduce the attack surface [9] of the services by not making them directly accessible from the outside, limiting the types of traffic that can reach it, and running on multiple diverse systems. Each request can use a different subset of diversified servers and the results are voted on making it more difficult for one attack to be universal. The servers are also hidden behind the firewall and proxies making information collection attacks more difficult as well.

**Description:**

**Details:** This technique aims to be a system capable of diagnosing issues, repairing itself, and reconfiguring itself in order to continue to provide a service in the event of attack. It consists of a firewall in the front that filters all traffic except http traffic. The traffic that gets through is fed to a network of proxy servers. These proxy servers communicate with the web servers. Each web server is a diversified system containing various architectures, operating systems, and software packages while providing the same content. The proxy systems are diversified in a similar manner but have been hardened to be further resistant to attack.

The proxy servers have the ability to take on differing roles. The proxy servers choose a leader and this leader handles all requests from the firewall. It is responsible for determining which subset of web servers should be used to process the client request. A different number of web servers can be chosen to process the request based upon how critical the service is or the current alert level of the system. It will also be responsible for making sure everything is load balanced so some servers are not overworked. One of the proxy servers is also chosen as an adjudicator that manages connections to the shared database if one is needed. It has the ability to filter out any suspicious looking SQL queries. Each proxy is capable of taking on one of these roles if something happens to the existing elected proxy. Each proxy also has an alert manager on it. Each proxy and server can be in a trustworthy state, a suspected state, or a corrupted state. The alert managers help decide what action should be taken in the event of an alert from any detection mechanism (described later). When something receives an alert, a vote is taken amongst the proxies to verify alert. An action is then taken depending on the role of the corrupted component. Each proxy is in a different administration domain to prevent one administrator from taking over all proxies. The leader proxy also has the ability to filter out any suspicious looking http requests.

This technique incorporates a number of different detection mechanisms. The first is an agreement protocol. This is a voting mechanism to determine if a server was corrupted. Each server processing the request sends a cryptographic hash of the response excluding the header back to the current leader. The majority response is then used to be sent back to the client. The proxies use this same voting protocol when an anomaly is detected in any of the systems to come to a

consensus on a countermeasure. The adjudicator proxy uses this protocol to verify SQL queries before executing them on the database as well.

The next detection mechanism used is a combination of network- and host-based intrusion-detection systems. Host-based intrusion detection systems are placed on every host and a network-based intrusion system is used to monitor the traffic between servers and proxies. Each web server is rebooted periodically from a read-only trustworthy source.

In between each two reboots of a system, another detection mechanism is implemented. This is a challenge-response protocol implemented by each proxy. A proxy can periodically send out a challenge to any other proxy or server about a file on that system. The response is checked against a precomputed response. There must be enough challenges generated to last between two reboots of a system.

The final detection mechanism implemented is a runtime verification of the proxies. This checks the behavior of each proxy during its execution. The system is modeled by a finite-state machine and the state is monitored. There are different models depending on the current role of the proxy. The proxy can both monitor its own behavior as well as the other proxies behavior.

**Entities Protected:** This technique is used to protect the availability and integrity of web services.

**Deployment:** This technique would be deployed in the overall network architecture.

**Execution Overhead:**

- Duplex and triplex regimes have 200% and 300% overhead plus additional overheads as follows:
  - For one server without database access and a 1 MB file, this added about a 31% overhead
  - For three servers without databases access and a 1 MB file, this added about a 33% overhead
- Database access time approximately doubled with this technique

**Memory Overhead:**

- None

**Network Overhead:**

- Additional network traffic generated by the additional servers

**Hardware Cost:**

- Additional platforms to host the additional variants

- Additional network infrastructure to support this platform configuration

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** A firewall able to effectively filter everything but HTTP traffic. The service being provided should produce the same results under normal conditions on all systems. The technique relies on a detection mechanism.

**Weaknesses:** An attacker could launch a large-scale attack that results in all the web servers rebooting due to detections causing a denial of service. In addition, this technique provides no protection against targeted attacks on one web server. Also, data leakage attacks are not protected.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☒ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** This technique could significantly increase the workload of the attacker. The attacker would need to create an attack that would work on an unknown majority number of diversified systems. An attacker may be able to leverage a script injection if it is not detected by any of the detection mechanisms. The attacker also cannot directly access the servers making it more difficult to do reconnaissance on those systems.

**Availability:** This technique was prototyped by the authors but is not publicly released.

**Additional Considerations:** The paper lacks specifics on the types of diversity or how that may impact security. It could be prohibitively expensive at large scale. The technique only handles http traffic. This method is limited and can only be applied to specific a system.

**Proposed Research:** One possible direction to look into would be to blacklist requests that caused a server to go into a bad state. Future requests that are on the blacklist could be blocked at the proxy. This would prevent a continuous denial of service attack using the same attack request continually. It may also be possible to combine this technique with other movement methods on the web servers. This could make them more resistant to a larger set of attacks and offer more detection mechanisms.

**Funding:** SRI International, DARPA

### 5.9 SCIT

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [115–117] does not detect any attacks but assumes the system is continually under attack. While this would not stop an attacker from injecting code, the minimal exposure time before cleaning a system would require a fast-acting exploit. This would also stop attackers from continuously persisting on these systems.

**Description:**

**Details:** The Self Cleansing Intrusion Tolerance (SCIT) technique aims to decrease the exposure time of a system by rotating it with copies. The copies that are not being used are cleaned and restored to a pristine state. Each system copy is implemented in a virtual environment. There is a separate system with a network attached memory utility that stores persistent short-term information or session data between the systems. The final component is a controller that manages the rotation of the systems and how long each system copy is exposed. The systems can be in one of four states. The first state is *active* where it is online and accepting/handling requests. The second state is *grace period* where it stops accepting new requests and finishes processing existing requests. The third state is *inactive* where it is taken offline to be restored. The final state is *live spare* where the system has been restored and is ready to become active. There can be either one active server at a time serving one service or multiple active servers serving multiple services. The latter would require additional algorithms to determine which systems could be easily brought down next. The systems are rotated on the order of minutes. The systems are on their own private virtual network and are not directly accessible from the internet. Connections to the systems are managed by a load balancing system.

**Entities Protected:** This technique protects servers by limiting their exposure time.

**Deployment:** This is a contained virtual environment and could be deployed on the servers.

**Execution Overhead:**

- Some unknown overhead due to virtualization
- Significant overhead for cleaning the VMs

**Memory Overhead:**

- None

**Network Overhead:**

- None – Self-contained virtual network

**Hardware Cost:**

- Additional hardware to support a virtualized environment

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☒ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☐ Exploit Development

☐ Attack Launch

☒ Persistence

**Interdependencies:** The technique requires a virtualization infrastructure in place. It also requires an automated re-imaging capability.

**Weaknesses:** The networked memory does not have any protection or integrity control. Since it is accessible via all systems, an attacker could attempt to quickly corrupt or change the contents of this storage. Another weakness is that every system is the same. If the attacker can find a working exploit against one system, it would work on all systems at once. In fact, since exploits work very fast, this technique provides little protection. The system also does not protect against data leakage attacks.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☒ Disable Movement

**Impact on Attackers:** This technique does not stop an attacker from exploiting a system. It decreases the amount of time an attacker has to accomplish their goal. This makes it more difficult to persist in the network. An attacker would have to compromise the load balancer or the networked memory system or quickly jump to new systems as the older ones are being re-imaged.

**Availability:** This technique was prototyped by the authors but is not publicly released; See `http://scitlabs.com/`

**Additional Considerations:** The technique provides little protection at a large cost. The attacker can always jump to the new platforms (since they are identical) and continue to persist. Moreover, the overhead to re-image the systems can be very large. The technique is also limited to specific servers that are almost stateless or the state can be persevered in the configuration (*e.g.*, DNS server).

**Proposed Research:** One direction for this technique would be to develop a better way to protect the network memory. If an attacker can continually change or corrupt the data, the effectiveness of this technique is significantly decreased. Another direction this technique could take would be to introduce diversity into the operating systems. Different architectures, operating systems, and servers that provide the same function could be used to increase the workload of the attacker. This would reduce the likelihood of an attack working across all systems. Also preserving the state beyond configuration files is a direction to explore.

**Funding:** Lockheed Martin, Virginia Center for Innovative Technologies

## 5.10 GENETIC ALGORITHMS FOR COMPUTER CONFIGURATIONS

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Scanning

**Details:** These techniques [118,119] have a long-term security goal and do not actively defend against or respond to attacks. The idea is that the evolution of configurations over time effect the lifetime of exploits and the varying configurations amongst systems helps prevent exploits from working against multiple machines. The evolving configurations make collecting information about a specific machine less reliable.

**Description:**

**Details:** This technique aims to find more secure configurations of systems over time using ideas from genetics. The security of a configuration is defined as the number and severity of security incidents reported while that configuration was active. A configuration can consist of many parameters in a system such as which desktop manager is being used or which remote login protocol is being used. The ideas that are being used from genetics include selection, crossover, and mutation. Selection involves selecting the best configurations based on their security score. Crossover involves taking two configurations and combining elements of each one to create a new configuration. Mutation involves randomly changing parts of a configuration to make it different from configurations on other systems. The goal is to create configurations with temporal and spatial diversity. Temporal diversity means the configuration of one machine changes over time. Spatial diversity means multiple computers do not have the same configuration at a given time.

How this process works is that every system starts with the same configuration. Since no other configurations exist yet, it is mutated to create a new configuration. If the resulting configuration is reasonable, it is set as the active configuration. After some time has passed, the security score is calculated for that configuration. If the configuration pool is not full, this configuration is added into this pool otherwise it replaces the worst configuration in the pool. The next iteration would involve taking the two best configurations from the pool, doing a crossover to create a new configuration, and applying a mutation to add some additional randomness to it. This new configuration then goes through the same process of seeing if it is a reasonable configuration, making it active, and calculating its security score. This process repeats over many iterations until ideally there are configurations that have no security incidents.

**Entities Protected:** This techniques aims to protect the operating system or servers by finding better configurations over time.

**Deployment:** This would be deployed on each system that has a similar purpose.

**Execution Overhead:**

- Unknown execution overhead due to reconfiguration

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

- ☐ Reconnaissance
- ☐ Access
- ☐ Exploit Development
- ☐ Attack Launch
- ☒ Persistence

**Interdependencies:** A detection mechanism or set of mechanisms to be able to calculate the security score is critical for this technique. It also assumes that the entire security posture of a system can be controlled via configuration.

**Weaknesses:** This can be deceptive and give a false sense of security. A configuration can be chosen as good but it could be the case where the system was just not under attack at the time. Another large weakness of this technique is that the security score is based on detected attacks and relies on the systems being constantly attacked. A stealthy attacker could still carry out their attack. It may also be possible to manipulate the configuration selection by causing detections on configurations the attacker does not want. Moreover, many important security aspects of a system cannot be controlled with a configuration (See [66]). Also, it can take a long time to converge to a good configuration. The technique does not protect against data leakage attacks or one-time attacks either.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☐ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** Depending on the configuration options being changed, this may have little effect on the effort of an attacker. In ideal conditions and with strong detection mechanisms, it could affect an attacker over time by making it more difficult for them to perform an attack.

**Availability:** This technique was prototyped by the author but is not publicly released.

**Additional Considerations:** Frequently changing the configurations can be impractical. Functionality of the system may break because of reconfiguration. It can take a long time to converge to a good configuration. Also systems evolve over time. New software could be install, old software removed, system patches applied, operating system upgraded, etc. All these changes will result in new or removed configuration options. Also the system crucially relies on a perfect detection and monitoring capability to operate correctly.

**Proposed Research:** A possible future direction for this project would be to expand it from simple configurations to actual software as well. It might try running a service with a specific server then later try running the same service with a different server that provides similar functionality. This does not fix the reliance on detection mechanisms. Additional aspects that would need to be investigated are how the configurations are combined to create new configurations, how configurations are randomized, and how long a configuration would need to be active to get a reasonable security score. Combining this technique with other movement techniques that provide more proactive protection against classes of attacks might provide more overall protection. The impact of configuration changes on the functionality of the system must be studied.

**Funding:** Pacific Northwest National Laboratory

### 5.11 MOVING ATTACK SURFACE FOR WEB SERVICES

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection, Control Injection, and Scanning

**Details:** This technique [120, 121] can help mitigate a variety of attacks. Since the service is being served randomly between systems with different frameworks, libraries, architectures, virtualization technologies, and layouts, it is more difficult to construct exploits that will work under every platform. The attacker also may not be able to predict when the system will change or which system the load balancer will choose to service the request. The fact that the system is changing constantly makes passively scanning and collecting information less useful.

**Description:**

**Details:** This technique employed diversification at different levels of a system and across many systems to create a varying attack surface across all the systems. The technique is composed of a frontend load balancer, a pool of virtual diversified systems, and a trustworthy controller. The idea is to have a subset of the virtual systems online at a time and the load balancer chooses which one will service a request. The trustworthy controller manages the virtual systems. Virtual systems can be in one of three states: online, graceful shutdown, or offline. In graceful shutdown, they do not accept new requests and finish processing any existing requests. In the offline mode, they are powered off. When a system is to come back online, it is restored to a pristine state. The virtual systems are randomly rotated in and out by the trustworthy controller. Three different mechanisms can trigger a system to be taken offline. The first mechanism can be event driven where anomalous events are detected or integrity checks fail. The second mechanism is a random selection where a system can be taken offline at any random time even if no anomalous activity is detected. The third mechanism is a maximum lifespan where a system is taken offline if the amount of time it has been online exceeds a limit in order to reduce exposure to attacks. This technique does not support services that require persistent state because they are wiped clean periodically. For this reason, the authors recommend a framework such a Representational State Transfer (REST) so the web servers can be stateless and still provide stateful services.

This diversification is done at the application level by using different implementations of the web framework being employed or different implementations of software that those web frameworks require. Diversification is done at the web server level by choosing different open source and commercial web servers. Additional diversification is achieved by hosting these web frameworks on the same web servers using different modules or technologies. Diversification is achieved at the operating system level by choosing from a variety of open source and commercial operating systems including Solaris, Windows, BSD flavors, and Linux flavors. Additional diversification is achieved by putting in a mix of 32-bit and 64-bit versions. Diversification is achieved at the virtualization level by using a mix of hypervisor-based and operating system level virtualization

technologies that support some form of checkpointing and restoration of systems. With all these levels of diversifications, the authors were able to come up with 1554 unique combinations.

**Entities Protected:** The primary function of this technique is to protect a web service but the diversification also helps protect the operating system as a whole.

**Deployment:** This technique would be deployed in the web services.

**Execution Overhead:**

- $K$ replicas impose at least $K$ times overhead in execution
- Some overhead imposed by running in a virtual environment

**Memory Overhead:**

- $K$ times memory used

**Network Overhead:**

- None

**Hardware Cost:**

- Hardware to support the various virtualization setups

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☒ Persistence

**Interdependencies:** If relying on the event-driven rotation, it is necessary to have good anomaly and integrity checking mechanisms in place. This technique also requires a web framework that supports running stateful services on stateless servers if stateful services are required. This is not straightforward to achieve for arbitrary web services.

**Weaknesses:** This technique does not protect against web-service logic bugs or failure to sanitize input. As a result, attacks like SQL injections could be leveraged if the service does not properly sanitize input or put other mitigations in place. The load balancer and trustworthy controller are both static machines and could be potential targets for an attacker. If an attacker can compromise the trustworthy controller, he or she could control or stop the system rotation process. If the system rotation process is not done quickly enough, the attacker may still be able to accomplish his or her goal if he or she is not trying to be persistent. It is also possible an attacker has a set of attacks that work only on certain combinations of software and the rotations of systems may eventually get to that configuration. More importantly, this technique does not protect against data-leakage attacks or attacks against one machine.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☒ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique would slow down an attack and make it more difficult for the attacker to exploit the machines but it would not stop him or her completely. An attacker still has the chance of exploiting a system and achieving his or her goal before the active system changes. An attacker could also try to leverage more advanced control-injection attacks that could work across multiple systems.

**Availability:** This technique was prototyped by the authors but is not publicly released. It is composed primarily of open source or commercial software.

**Additional Considerations:** The impact of diversity on identifying attacks is unknown. Having a large number of diversified systems would increase the management and maintenance complexity. This technique is only limited to a web server. Extending the technique to a generic service can be very difficult.

**Proposed Research:** This technique could potentially be combined with additional internal operating system movement techniques to slow down the attackers further giving the system additional time to migrate systems. Ensuring that the trustworthy controller is isolated and the virtual systems are not able to manipulate it would also be a worthwhile avenue to explore. The impact of various types of diversity on attacks must also be studied.

**Funding:** Unknown

## 5.12 LIGHTWEIGHT PORTABLE SECURITY

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection and Control Injection

**Details:** This technique [122] helps mitigates persistent threats on a system by ensuring the operating system boots into a clean and known-good state. The system can be rebooted in between sessions to return the system to a good state removing any infections incurred during the last session.

**Description:**

**Details:** The Lightweight Portable Security technique involves booting a system into an isolated and minimal operating system. This operating system resides only in the memory of the system and does not access any internal persistent storage devices. The operating system is on a read-only bootable device or media ensuring that it cannot be corrupted or modified in a malicious matter. The operating system is built off of the Linux operating system and includes a basic set of applications such as a web browser, smart card middleware (such as the Department of Defense Common Access Card or US Government Personal Identity Verification card), encryption software, file browser, image viewer, PDF viewer, text editor, remote desktop software, and SSH client. It also includes the ability to use external storage devices such as USB hard drives and memory sticks. The public editions of this technique allow a person to browse the Internet without putting their local machine at risk. The deluxe public edition has all the software of the regular public edition with the inclusion of additional software such as office software. The Remote Access version of this technique is meant to connect to enterprise networks and use internal network resources and it is customized for each particular customer.

**Entities Protected:** This technique protects a user session by booting into a known good and clean state. There are two primary use cases for this technique. If a person wants to browse untrusted websites and wants to protect his local computer, he can boot one of the public editions of this technique. A person might not trust the local computer and he wants to protect his online session. In this case, he can boot from one of the editions of this technique and do activities like online banking or connect to his work network securely without worrying about the local machine assuming the hardware/firmware is trustworthy.

**Deployment:** This technique would be deployed on a generic machine by booting from a read-only media.

**Execution Overhead:**

- Extra time required for re-booting into another OS

**Memory Overhead:**

- Unknown memory overhead due to removal of hard drive

**Network Overhead:**

- Some overhead incurred if connecting through trusted networks

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification required)

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☐ Exploit Development

☐ Attack Launch

☒ Persistence

**Interdependencies:** The technique assumes that secure and lightweight versions of the OSes are available.

**Weaknesses:** This technique does not protect against compromised hardware in the system or hardware connected externally to the system. An attacker could re-flash firmware on the machine and persist. It is also possible for an untrusted external hardware device, such as a USB hard drive or memory stick, is connected to the local machine. Since the technique supports using external hardware, a new session may not be trusted if these external devices are mounted automatically. More importantly, the technique does not provide any protection after booting into a new OS. The sessions can still be compromised and information can still leak. The technique relies on rebooting after performing any important operation or for performing potentially dangerous actions (browsing an unknown website).

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique removes the persistence of attacks. It makes it harder for an attacker to remain on the system. Once a new session is initiated, any malicious code that was placed on the system will be removed.

**Availability:** This technique is available in three different editions. There is a public edition, public-deluxe edition, and remote access edition. The first two editions are free to download and use but the third edition must be requested from the agency. See `http://www.spi.dod.mil/lipose.htm`

**Additional Considerations:** Requires booting a system each time it needs to be used. It can have a very large overhead due to rebooting. In many cases, it is difficult to distinguish between benign and potentially dangerous actions. It must connect external devices for local persistent storage. The OS runs completely in memory so the host would need an adequate amount in the local machine.

**Proposed Research:** This technique does provide reasonable protection from persistent threats but it does not address all the locations a persistent threat could reside. The hardware firmware inside of the host machine could have been tampered with in a malicious manner. An interesting research direction might be to see if it is possible to leverage trusted hardware technologies to verify hardware has not been tampered with as well. If the user intends to create a secure session because he does not trust the local machine, it would also be good to look into making sure potential untrusted or malicious external devices connected to the local machine are not automatically mounted into the trusted environment. One can also look into making reboots faster and more efficient.

**Funding:** Air Force Research Laboratory

## 5.13 NEW CACHE DESIGNS FOR THWARTING SOFTWARE CACHE-BASED SIDE-CHANNEL ATTACKS

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Data Leakage Attacks

**Details:** This technique [123] aims to prevent a process from learning privileged information about another process running on the same machine via observations of the cache behavior. The attacker-controlled process is an unprivileged, user process and is limited to performing operations normally permitted to user processes. The attacker does not exploit software vulnerabilities or perform physical attacks.

**Description:**

**Details:** Several side-channel cache attacks enable an unprivileged process to deduce values, such as encryption keys, by filling up the cache with known values and monitoring for when a value is evicted by the victim process. The authors propose two approaches for combating these cache side-channel attacks: partitioning the cache for different processes to avoid interference, and using randomization during the eviction process. The randomization technique adds a layer of indirection to the cache structure. If the cache line to be evicted belongs to another process, the cache will randomly select a different set from which to evict. This prevents the attacker from determining which value was evicted.

**Entities Protected:** This technique protects a user application on a machine from other applications running on the same machine.

**Deployment:** The goal of the FY13 funding is to produce a hardware chip that would replace the cache hardware in existing machines.

**Execution Overhead:**

- Less than 2% performance overhead on average

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- Cache must be replace with a new chip

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☒ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** Designed for existing general purpose computing architectures.

**Weaknesses:** This technique does not protect against physical attacks or provide defense against software vulnerabilities. Despite the less than 2% reported performance overhead, any overhead may be a significant hurdle to adoption due to the relative scarcity of the attacks for which the technique protects against.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** In order to achieve data leakage, attackers must exploit vulnerabilities in the target software or have physical access to the system.

**Availability:** Not publicly available.

**Additional Considerations:** Cost and performance will be significant considerations for this technique. The goal of the FY13 work is to provide no performance hit via an improved hardware chip; if this is achieved at negligible cost, this could be a promising technique for next-generation systems.

**Proposed Research:** Implementing randomization ideas into hardware has received comparatively less attention in the community when compared with software-based randomization techniques. Hardware-based randomizations in general could be an interesting area for further research.

**Funding:** 2007 work funded by DARPA and NSF Cybertrust CNS- 0430487 and CNS-0636808, DoD and Intel. 2013 work funded by DHS BAA 11-02 TTA12.

## 5.14 NOMAD: MITIGATING ARBITRARY CLOUD SIDE CHANNELS VIA PROVIDER-ASSISTED MIGRATION

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Data Leakage Attacks

**Details:** This technique [124] can mitigate arbitrary side-channel based data leakage between collocated VMs in a cloud environment. The VM is migrated periodically to different physical machines, in order to bound the amount of information that can potentially leak to malicious VMs sharing hardware with the victim. The threat model considers an adversary with multiple VMs whose placement across physical machines is unknown to the provider or victim. It also considers victims who use replication of information across VMs. Under these conditions, it can provide a mathematical bound on the amount of information leaked assuming a known or estimated leakage rate. Since this technique does not try to stop specific side-channels (*e.g.*, cache), it is applicable to both known and yet-to-be-discovered data leakage based on shared hardware.

**Description:**

**Details:** Nomad is a security-conscious VM migration algorithm for defending against side-channel data-leakage attacks mounted by collocated virtual machines. It is run by the cloud provider, and does not require any changes to the VMs themselves. Nomad is designed to mitigate all possible hardware side-channels between VMs, by periodically migrating VMs to a new physical machine. Nomad operates by providing an API to clients that allow them to specify constraints on migration in terms of required availability. It then runs a placement algorithm that minimizes the global information leakage across all pairs of VMs in that cloud (by trying to minimize the time they are co-resident) while also minimizing the number of VMs that have to be moved in order to maintain availability. This algorithm is rerun periodically, resulting in a new assignment of VMs to physical machines.

The placement algorithm can be configured to minimize information leakage for one of four scenarios based on features of the VMs and of the attacker. VMs may be either replicated or not. Replication means that sensitive information is copied across victim VMs, such as a web server with private database records. This could enable an attacker to learn sensitive information from any one of several VMs, increasing the attack surface. Attackers may be either collaborating or not. Collaborating attackers can work together if a victim moves from one collaborator to another. An example of this is one attacker learning the first half of the bits of a cryptographic key, and a second attacker learning the latter half, then sharing with one another. These two features (replication and collaboration) combine into four possible scenarios, with the combination of VM replication and collaborating attackers offering the most difficult security challenge. Note that the placement algorithm cannot optimize for more than one such scenario at a time.

**Entities Protected:** Virtual machines running inside of a cloud environment with a trusted provider.

**Deployment:** This technique could be deployed by any cloud provider. The only changes required are an update to the VM migration algorithm. No changes to the VMs themselves are necessary.

**Execution Overhead:**

- The overhead of Nomad has two components: overhead suffered by the provider in computing VM placement, and overhead suffered by the VMs by migrating them.
- The provider overhead is very minimal, with clusters containing thousands of virtual machines incurring less than one second computation time.
- The VM overhead is specified by the client VMs as a minimum availability. Nomad will never violate this, but security may be reduced if a VM has to be highly available (since it cannot be moved as frequently).

**Memory Overhead:**

- The authors do not report memory overhead for the provider.

**Network Overhead:**

- The authors do not report network overhead incurred by migration, but acknowledge it may be a concern

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** Nomad relies upon an existing VM migration system being present in the cloud on which it is deployed.

**Weaknesses:** This technique has several weaknesses, though none render it unusable. First, Nomad can only optimize security for one scenario at a time, i.e. one combination of (replicated/unreplicated and collaborating/non-collaborating), even though differing users of the cloud may have different workload types and threat models. This means that at any time, at least one class of users might be suffering from reduced performance or security. Second, Nomad only provides a quantifiable benefit if the provider is aware of the leakage rate of information from VMs. This may be challenging to estimate, especially if the adversary is aware of high-bandwidth side channels that the provider is not aware of. Third, an attacker who can mount a Sybil attack (i.e. acquire many VMs all under one attacker's control) may create a high probability of always being collocated with a victim before and after migration. Finally, Nomad optimizes global security. A specific VM might suffer poor security, if in so doing it increased the security of all other VMs more.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique could significantly hinder attackers trying to leak sensitive data across VMs, such as cryptographic keys. The attacker would either have to have a large number of virtual machines, or use a side-channel of sufficiently high bandwidth to overcome the time limits imposed by Nomad's migration system.

**Availability:** No code publicly available

**Additional Considerations:** None

**Proposed Research:** Nomad's primary unknown overhead is in network bandwidth consumed by migration. This could potentially be evaluated and mitigated by integrating Nomad with a Software-Defined Networking component that could reserve and release bandwidth in advance (since Nomad's movement is predictable). Additionally, a technique to bound the amount of harm individual VMs can suffer due to global optimization would be useful.

**Funding:** National Science Foundation

### 5.15 MULTIPLE OS ROTATIONAL ENVIRONMENT

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection, Control Injection, Scanning

**Details:** This technique [125], called MORE, is designed to protect against a remote adversary attempting to exploit the operating system of a vulnerable service, such as a web server, but not the service itself. The adversary is assumed to be scanning for vulnerabilities, as well as potentially using zero-day exploits once the set of open services is identified. MORE seeks to mitigate this threat by periodically changing the operating system being used to host the service. If done rapidly enough, rotating the operating system will cause an attacker's vulnerability scans to become stale and no longer usable.

**Description:**

**Details:** MORE hosts a web service from a set of active servers, each of which runs a different operating system (though the authors restricted this to different Linux distributions). All of these servers are supported by a common database backend. The technique uses a pool of active IP addresses and a single "spare" IP address. A front-end load balancer directs incoming requests randomly to each active IP address. Periodically, a VM is rotated out of the set of active IP addresses and assigned the spare address. It is then inspected for signs of intrusion, and may be restored to a known-good state. At the next rotation period it will be assigned an active IP address, and a previously active VM will be rotated into the spare address for inspection.

**Entities Protected:** Virtual machines hosting potentially vulnerable servers

**Deployment:** This technique could be deployed by any system administrator. It uses pre-existing technology and common system architectures.

**Execution Overhead:**

- No analysis of overhead was available

**Memory Overhead:**

- No analysis of overhead was available

**Network Overhead:**

- No analysis of overhead was available

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☒ Persistence

**Interdependencies:** MORE relies on strong intrusion detection systems in order to detect compromised servers that have been rotated to the spare IP address. It also requires that the service being hosted can be installed and run on all the operating systems used to provide diversity.

**Weaknesses:** This technique has several weaknesses. It does not protect the actual service being hosted, which remains vulnerable to both application-layer attacks (*e.g.*, SQL injection) and exploitation of memory corruption vulnerabilities. The fact that rotation is periodic provides a time window during which attackers can scan and attack the system as normal, and increasing the rotation rate will likely have performance implications. Additionally, an intelligent attacker can simply wait for a scanned VM to come back into rotation. This means that the attack surface is actually increased. Once all operating systems have been identified through scanning, an attacker only needs to find a vulnerability in any one operating system, then simply wait for that VM to become reachable again.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** This technique could increase the time it takes an attacker to identify and exploit an operating system. However, as discussed above, it may actually help the attacker by increasing the variety of targets the attacker can choose from.

**Availability:** No code publicly available

**Additional Considerations:** Diversifying the operating systems used to host a service will increase management and maintenance complexity.

**Proposed Research:** This technique could potentially be combined with either static or dynamic application layer diversity, such as through use of a multi-compiler. It could also benefit from automated intrusion detection and cleaning mechanisms.

**Funding:** Department of Energy

### 5.16 DYNAMIC APPLICATION ROTATION ENVIRONMENT FOR MOVING TARGET DEFENSE

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Code Injection, Control Injection, Scanning

**Details:** This technique [126], called DARE, considers a remote attacker attempting to exploit a vulnerable web service listening on an open port. This technique is designed to protect against a remote adversary attempting to exploit the operating system of a vulnerable service, such as a web server, but not the service itself. The adversary is assumed to be scanning for vulnerabilities, as well as potentially using zero-day exploits once a web server is identified. DARE seeks to mitigate this threat by periodically changing the server being used to host the service. Specifically, they consider a web server hosting an Internet-facing web page. If done rapidly enough, rotating the server may cause an attacker's vulnerability scans to become stale and no longer usable.

**Description:**

**Details:** DARE hosts a set of diversified web servers, each using a different software package to host the same website. A load balancer front-end randomly directs incoming requests to one of the servers, based on a random but bounded service interval, before switching to another server. The authors specifically consider Apache and Nginx. Each is configured to listen only on localhost, with all requests first passing through the load balancer, which is the only Internet-accessible component.

**Entities Protected:** Internet-accessible servers, such as web servers

**Deployment:** This technique could be deployed by any system administrator. It uses pre-existing technology and common system architectures.

**Execution Overhead:**

- The authors reported that overhead was negligible

**Memory Overhead:**

- The authors do not report memory overhead

**Network Overhead:**

- The authors perform only preliminary measurement of network overhead, but note that routing table updates can cause spikes in request latency of several hundred milliseconds

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** DARE relies on the service it is hosting to be stateless, which is nontrivial to implement for many services beyond static websites.

**Weaknesses:** This technique has several weaknesses. It only protects the specific service being diversified, not the underlying operating system or support software. The fact that request redirection is semi-periodic provides a time window during which attackers can scan and attack the system as normal, and increasing the redirection rate will likely have performance implications for request latency. Additionally, an intelligent attacker can simply wait for a scanned and identified server to come back into service (and can confirm this with subsequent scans). This means that the attack surface is actually increased. Once all server software packages have been identified through scanning, an attacker only needs to find a vulnerability in any one server, then simply wait for that VM to become reachable again.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** This technique could increase the time it takes an attacker to identify and exploit a web service. However, as discussed above, it may actually help the attacker by increasing the variety of targets the attacker can choose from.

**Availability:** No code publicly available

**Additional Considerations:** Diversifying the web servers used to host a service will increase management and maintenance complexity. Additionally, stateful services will have to be made stateless (potentially by backing them with a stateful database). This is a potentially complex and expensive architectural change.

**Proposed Research:** This technique would benefit from the ability to automatically generate diversity among web servers, as well as the ability to re-diversify on a periodic or random basis (*e.g.*, via compiler diversity). This would prevent attackers from predicting or waiting for previously-identified servers.

**Funding:** Department of Energy

### 5.17 SCHEDULER-BASED DEFENSES AGAINST CROSS-VM SIDE-CHANNELS

**Defense Category:** Dynamic Platforms

**Defense Subcategory:**

**Threat Model:**

**Attack Techniques Mitigated:** Data Leakage Attacks

**Details:** This technique [127] can mitigate side-channel-based data leakage between collocated VMs in a cloud environment via scheduler-based soft isolation. By limiting the frequency of preemptions by other VMs sharing the same processor, the technique disrupts Prime+Probe side channel attacks. The threat model considers an adversary that shares a processor with the victim machine, such as in cloud environments. The shared hardware leaks, among other data, timing information about memory accesses by the victim. This can be used to leak, *e.g.*, cryptographic keys via a method called Prime+Probe. The attacker primes a shared cache by filling it with entries via accesses to a fixed set of addresses. It then yields execution to the victim, whose own computation cause the attacker's cache entries to be evicted. The attacker then preempts the victim as quickly as possible, regaining control of execution. It accesses the address set again, and by measuring the time to access memory can determine which entries were evicted by the victim. This leaks informations about which memory addresses were accessed by the victim.

**Description:**

**Details:** This technique takes advantage of a scheduling feature in the Xen hypervisor that allows for rate limiting of VM preventions, called Minimum Run Time (MRT) scheduling. The MRT parameter determines a minimum length of time that any virtual CPU (VCPU) can run on a physical CPU (PCPU) before being preempted by the hypervisor and switched out to allow another VM to execute. This technique leverages MRT scheduling to impose soft isolation and disrupt Prime+Probe side channel attacks. The authors experimentally confirm that an MRT longer than the compute time of a sensitive computation (such as encryption) completely removes the ability of an attacker to measure intermediate computation states via priming and probing of the victim. For the ElGamal algorithm on the authors' testbed, this computation time is 2 ms, and an MRT of 5 ms is sufficient to disrupt any cache-based side channel.

The authors augment the MRT defense with core-state cleansing in order to extend protection to interactive workloads that may voluntarily yield execution before the MRT expires. This mechanism operates by executing a sequence of machine code instructions that overwrite all lines of the instruction, data, and branch prediction caches. The scheduler runs the cleansing sequence whenever an interactive process yields the VCPU, prior to the next VM beginning its execution.

**Entities Protected:** Virtual machines running in a cloud environment

**Deployment:** This technique can be deployed easily on any cloud platform using Type 1 hypervisors, specifically Xen.

**Execution Overhead:**

- Execution latency scales proportionally with MRT for computationally intensive workloads, which may be unacceptable for latency-sensitive workloads. CPU-intensive batch workloads are only minimally affected, and often benefit from high MRT. This is not surprising, as it was the original motivation for providing MRT scheduling options.

**Memory Overhead:**

- None

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☒ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique relies on scheduling features of the Xen hypervisor. More generally, any Type 1 hypervisor will suffice as long as Minimum Run Time scheduling can be implemented.

**Weaknesses:** This technique only mitigates side channels based on shared access to a common cache. Other side channels (*e.g.*, disk access time or power analysis) feasible. Additionally, if the MRT is ever set lower than the execution time of a sensitive computation, noisy but usable information may leak to an attacker.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique could significantly hinder attackers trying to exploit cache-based side channels to leak sensitive data across VMs, such as cryptographic keys. The attacker would have to rely on nonce-cache based side channels, which may entail physical access or reliance on indirect measurements of computation state such as disk access.

**Availability:** No code publicly available

**Additional Considerations:** None

**Proposed Research:** This technique is currently specific to cache-based attackers. However, the idea of soft isolation may be applicable to other side channels of interest, such as power usage or I/O.

**Funding:** National Science Foundation

### 5.18 DÜPPEL

**Defense Category:** Dynamic Platforms

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Data Leakage Attacks

**Details:** This technique [128] can mitigate data leakage, via L1 and L2 cache-based timing side-channels, between collocated VMs in a cloud environment. Noise is injected into instruction access times via a kernel module run by the defending virtual machine. The threat model considers an adversary virtual machine that shares at least one CPU core with the victim. Specifically, it assumes the adversary shares an L1 or L2 cache with the victim. Other side channels, such as I/O or power, are considered out of scope. There is some discussion of adapting Düppel to other cache-based side channels (*i.e.*, branch prediction), but this is considered future work. Additionally, Düppel does not defend systems using SMT, or protect the shared L3 cache.

**Description:**

**Details:** This technique periodically executes a cache cleansing procedure, which evicts all cache entries. This effectively erases any cache timing patterns caused by the target VM's computation, as post-eviction, all memory addresses will have equivalent access times. Two optimizations are used to determine how frequently the cleansing procedure should be executed.

The first uses dual modes of operation: sentinel and battle mode. In sentinel mode, cache cleansing happens infrequently, and imposes minimal overhead. Düppel monitors the preemption rate of the protected VM (*i.e.*, how frequently the VM is suspended while another VM executes). If the number of preemptions exceeds 10 per millisecond, Düppel assumes a side-channel attack (which relies on rapid preemption) is in progress and switches to battle mode. In this state, the cache is cleansed much more frequently, leading to increased overhead but higher noise induced on the side channel.

The second optimization limits protection to critical code regions. In order to avoid requiring source code modifications to applications, Düppel makes sensitive code pages in memory as non-executable. When the program attempts to run that code, the resulting page fault is used to make the memory readable and initiate a cache cleanse. This does require a user to enumerate sensitive code regions and manually configure Düppel to protect them, however.

**Entities Protected:** Virtual machines running in a cloud environment

**Deployment:** This technique can be employed by loading the Düppel kernel module into the VM to be protected. No hypervisor or cloud infrastructure changes are necessary.

**Execution Overhead:**

- The authors report a worst-case overhead of 7%, and an average of 4%.

**Memory Overhead:**

- None, beyond loading the kernel module

**Network Overhead:**

- None

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** Düppel requires a Type 1 hypervisor to be running. The current implementation also specifically requires Xen. Its preemption detection mechanism uses a shared datastructure provided by Xen that records the current system time.

**Weaknesses:** Düppel has some weaknesses associated with preemption and mode switching. First, because it runs in the VM it is protecting, Düppel's cleaning operation may itself be preempted by the hypervisor, resulting in an incompletely cleansed cache that contains data useful to attackers. Second, Düppel uses configurable timing intervals to schedule cleansing. Unless it runs after every instruction (which is prohibitively expensive), an attacker can still leak information by probing the cache between cleansing operations. Finally, Düppel's preemption detection threshold is arbitrary. An attacker could avoid triggering Düppel to switch from sentinel to battle mode by slowing the rate of attack. This causes loss of precision, but is likely preferable to triggering the defense.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☒ Limit Movement

☐ Disable Movement

**Impact on Attackers:** Düppel mitigates the amount of information leaked to attackers via L1 and L2 cache side channels. The authors trained an SVM classifier to identify sensitive instructions. Without Düppel the classifier has 90% accuracy, and with Düppel the accuracy dropped to 38%. This indicates that the defense limits, but does not eliminate, the amount of information that can be leaked.

**Availability:** No code publicly available

**Additional Considerations:** None

**Proposed Research:** Düppel relies on timing intervals to trigger cache cleaning. It would be useful to investigate synchronizing cleansing with possible attack actions, such as preemption. This would provide stronger guarantees about data leakage, and avoid some of the weaknesses relating to Düppel itself be preempted. It would also be valuable to extend this approach to other cache-based side channels, such as the branch prediction cache.

**Funding:** National Science Foundation

# 6. DYNAMIC NETWORKS

## 6.1 DYNAT

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Scanning, Resource, Spoofing, and Data Leakage

**Details:** This technique [129, 130] assumes the hosts and entities employing this technique are safe. It can help mitigate scanning attacks by obfuscating various parts of network packet headers but not the payload of the packets. Depending on the placement of the obfuscator, it could be used to combat some resource attacks like denial of service attacks. If the attacker is sending a flood of packets, the protected packet fields would be unencrypted and produce random values that would likely result in them not hitting the intended service. This same property would also increase the likelihood of detecting anomalies. This technique would also increase the difficulty of performing some spoofing attacks. It would be more difficult for an attacker to capture some traffic and replay it back to the service because of the changing obfuscation keys and uncertainty about how the network is currently mapped.

**Description:**

**Details:** Dynamic Network Address Translation (DYNAT) is a protocol-obfuscation technique. The idea is to randomize parts of a network packet header. This randomization can make it more difficult to determine what is happening on a network, who is communicating with whom, what services are being used, and where the important systems are located depending on how the technique is deployed. Some parts that can be scrambled include the Media Access Control (MAC) source and destination address, Internet Protocol (IP) source and destination address, IP Type of Service (TOS) field, Transmission Control Protocol (TCP) source and destination port, TCP sequence numbers, TCP window size, and the User Datagram Protocol (UDP) source and destination port. Ideally, the randomization is done with a strong cryptographic hashing scheme or encryption. The key can be changed on a clock-based scheme or via properties in the network such as packets sent. The key used to scramble can be generated via static means on each host, it can be split to be partially static and partially locally or externally dynamic, or it can be fully locally or externally dynamic.

**Entities Protected:** This technique aims to protect the network traffic as it is traveling between systems.

**Deployment:** This technique can have a number of different deployment scenarios depending on the level of protection needed. It can be deployed to workstations, servers, routers, and gateways. This could be used to protect switched Local Area Network (LAN) segments, contention-based LAN segments, LAN-to-LAN connections (local router connections), Gateway-to-Gateway connections (networks separated by the internet or long range connection), or a combination of LAN segments and gateway connections.

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- Depending on the deployment and fields obfuscated, the network overhead can be significant. For instance, if using this on a switched network and obfuscating the MAC address, this could cause the switches to fill up their memory and cause significantly more Address Resolution Protocol (ARP) traffic to determine which switch port to route packets through next.

**Hardware Cost:**

- Additional hardware may be required to handle the routing overhead.

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☒ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** To make this technique more effective, it should be used in combination with a packet payload encryption mechanism. Possibilities might include Secure Sockets Layer (SSL) or the IP Security (IPSec) protocol. Another mechanism needed is a reasonably strong encryption mechanism for the protocol obfuscation. A mechanism to generate new keys securely across all the participating systems is also necessary.

**Weaknesses:** The use of other networking protocols can reduce the effectiveness of this technique. Additional information is added to the packet headers with protocols like Multiple Protocol Layer Switching (MPLS) or using static Virtual Local Area Networks (VLAN). This additional information cannot be obfuscated and would leak additional information about what is going on inside the network. This technique does not do anything to change packet sizes, vary packet timing, or use dummy packets so it is susceptible to traffic analysis. More importantly, this technique only limits reachability. For services that can be reached from outside, this technique offers no protection.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☒ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique increases the workload for an attacker but does not necessarily stop them from collecting the information they need. Traffic analysis could still be used to profile types of traffic or the payload of the packets could be analyzed to collect information about the network.

**Availability:** This was prototyped by the original authors but is not publicly released.

**Additional Considerations:**

- This technique can severely limit a server's functionality because it cannot be reached from outside.
- Depending on the placement of these obfuscators, it could have adverse effects on other network equipment. For example, placing them behind routers or gateways may inhibit that device's ability to do traffic filtering.
- Depending on the fields obfuscated and the placement of the obfuscators, it could have adverse effects on other network equipment. For example, if MAC address obfuscation is being used, it could break port locking on switches if the MAC address is changing constantly due to obfuscation key rotation.
- Depending on the fields obfuscated, it could have adverse effects on other network protocols. For example, if MAC address obfuscation is being used, it could break dynamic Virtual Local Area Networks (VLAN).

**Proposed Research:** This technique could be expanded to harden it against traffic-analysis techniques. The obfuscators could be modified to include additional scrambling. This could include varying the timing of packets are sent from the system, inserting extra padding into the packets to vary packet size, and sending out dummy packets. Payload encryption is not currently a part of this technique and it increases the effectiveness of the technique by not allowing the attacker to analyze the content of the packets. More research would be needed to determine if there are more cases of special protocols leaking information making this technique less effective.

**Funding:** Sandia

### 6.2 REVERE

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource, Spoofing, and Data Leakage

**Details:** This technique [131] can help protect against a couple of classes of attacks to some degree. It helps protect against resource attacks like denial of service or manipulating content on the network. The effects of denial of service attacks are mitigated by the distributed and well-connected nature of the overlay network. An attacker would need to be able to flood potentially many thousands of machines simultaneously. This technique helps protect against content manipulation by using digital signatures on the content that it is distributing. This allows every node in the network to verify the content assuming the signature has not been compromised. This technique also helps protect against some spoofing attacks like man-in-the-middle, traffic replay, and impersonation attacks. This would help mitigate man-in-the-middle attacks by using strong authentication and trust relationships between each node in the network. Content replay attacks are mitigated by dropping duplicate content at each node. Impersonation attacks are also mitigated by the use of public key cryptography and digital signatures.

**Description:**

**Details:** Revere is a technique that involves creating an open overlay. An overlay network is an example of a dynamic network in that it can change paths, reconfigure, and respond to links or nodes going down dynamically. The network consists of a central distribution center that is the root of the network and nodes, or clients, receiving the content from the distribution center. Each node in the network can be a parent or a child. A parent can have multiple parents and multiple children. When a new node wants to join the network, it determines the fastest parent that it can attach to and performs a handshake with that parent to see if it will accept the new node. Once a node has found a parent, it then seeks out other additional parents to increase its resiliency.

Security is accomplished in this overlay by the distribution center digitally signing the content it is pushing out. Each node in the network can verify the signature of the content before using it and passing it on to its children. If the authenticity of an item is in question, it can be pushed back up to the node's parents and eventually the distribution center to be verified. Security can also exist between the parent and child nodes. Each node can support some set of authentication methods and the child can negotiate a method with the parent. Security appliances or authorities can also be employed for this task such as a Certificate Authority. Each node can have its own set of rules to determine if it should trust a parent or a child when they are negotiating.

Reliability is accomplished by the many-to-many relationships between the nodes. This provides many paths for content to be delivered and duplicate items are dropped at a node. Each node employs a heartbeat-type message between its parents and children to determine if they are still online. If a child does not receive a heartbeat from a parent in a certain amount of time, it

will assume that parent is gone and not use it anymore. Each parent is also capable of sending a message to tell a child that it is no longer usable.

Fast delivery is accomplished by each node maintaining the fastest path back to the distribution center. Each child has a Parent Path Vector (PPV) that is the fastest path back to the distribution center, which includes that parent. It also maintains a Node Path Vector (NPV) that is the fastest of the PPVs. If a parent that is part of the NPV goes down then the next fastest of the PPVs is chosen to be the new NPV. The speed of a link can be calculated at the child by analyzing the timestamps of the periodic heartbeat messages. The mesh-like distribution of nodes also helps push content out quickly.

The technique was prototyped as a Java client and tested up to 3000 nodes. Their testing showed that an update could reach all nodes in less than one second on average. They projected that an update could reach every node on a network of one hundred million nodes in less than four seconds.

**Entities Protected:** This technique protects the integrity and availability of content delivered over a network.

**Deployment:** This technique would be deployed as a client on a system that wishes to participate in the overlay.

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- The control messages passed between nodes does cause extra traffic on the network. Reconfiguration and routing can impose unknown network overheads.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification required)

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** This technique relies on good authentication mechanisms between nodes, good rules to determine if a node is trustworthy, and the security of the distribution centers.

**Weaknesses:** The security of the updates relies on the security of the distribution center. The authors mention that there are backup private keys available to use if one is compromised but, if an attacker is able to compromise one, it is not unreasonable to conclude the attacker could compromise the backups as well. This would allow the attacker to masquerade as the distribution center and push out fake updates, pollute the update repositories, or do other tampering of the content. The

node trust mechanism suffers from a similar problem. If keys are being used to sign messages to determine the authenticity and trustworthiness of a node, an attacker could have compromised a previously trusted host and use their identity. Also more importantly, the technique is focused on protecting reachability, if the machine can be reached from outside the overlay network, this technique does not provide any protection.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☐ Limit Movement
- ☐ Disable Movement

**Impact on Attackers:** The amount of impact this has on an attacker depends on the attacker's goals. If the attacker intends to poison the network with malicious or corrupted updates then the impact is correlated to the difficultly of compromising the distribution center and the private keys. If the attacker were attempting to bring down the network, the increase in difficultly would be correlated to the size of the network. The larger and more connected the network is, the more difficult it would be for an attacker to disrupt it as a whole. However, this technique does not provide protection for individual hosts.

**Availability:** This technique was prototyped by the authors but is not publicly available.

**Additional Considerations:** Some aspects of the paper are left very vague. Having a large trusted network or authentication between all nodes in the network is a good idea but if the network is spread across the world, how is setup for a new node wanting to join the network done? It discusses setting trust rules for a node but it is not clear what such a rule would entail or how a system could determine if another node is truly trustworthy simply by a handshake request.

**Proposed Research:** A dynamic network solution combined with other host-protection techniques must be explored.

**Funding:** Unknown

### 6.3 RITAS

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource, Exploitation of Privilege/Trust, Scanning

**Details:** This technique [132] is meant to impede an attacker from manipulating messages on the network or taking a service offline. The proposed protocols allow various processes to reach agreement on a message while accounting for a certain threshold of bad processes. Running multiple instances of processes also creates more redundancy in the service.

**Description:**

**Details:** Randomized Intrusion-Tolerant Asynchronous Services (RITAS) is a technique that builds a set of fault-tolerant consensus-based protocols on top of Transmission Control Protocol (TCP) and the Internet Protocol Security (IPSec) protocol. TCP provides a reliable channel and IPSec provides integrity to the data being transmitted. This technique is to be used among a set of $n$ processes. A process is considered corrupt if it does not follow its protocol until termination. This technique can handle at most $f = \lfloor \frac{(n-1)}{3} \rfloor$ corrupt processes. There are no assumptions about bounds on processing times or communication delays. The processes are assumed to be fully connected and each pair of processes share a secret key.

The first protocol is Reliable Broadcast. This protocol ensures that all correct processes deliver the same message and, if the sender is correct, the message is delivered. The next protocol is Echo Broadcast. It is a more efficient and less powerful version of the first protocol. It does not guarantee all processes will deliver a message if the sender is corrupt. The first consensus protocol is Binary Consensus. It builds upon Reliable Broadcast and allows processes to agree on a binary value (either one or zero). It is the only protocol of this technique that includes randomization if a consensus cannot be made. The next consensus protocol is Multi-valued Consensus. This allows the processes to agree on arbitrary length values and builds on top of Reliable Broadcast, Echo Broadcast, and Binary Consensus. The next consensus protocol is Vector Consensus. It allows the processes to agree on a subset of proposed values. It builds on the Reliable Broadcast and Multi-valued Consensus protocols. This ensures that each process decides on the list of values of size equal to the number of processes. Each element of the list corresponds to a process (element one of the list is the value of process one and so on). This ensures that each element of the list is either the value proposed by that process or the default value and at least $f + 1$ elements were proposed by correct processes. The final protocol is the Atomic Broadcast protocol. This protocol builds on Reliable Broadcast and Multi-valued Consensus. This protocol ensures that each message is delivered reliably and in the same order to all processes.

Using randomization, this technique implements a dynamic network that is capable of guaranteed delivery given limited number of malicious nodes.

**Entities Protected:** This technique protects the information returned from a service by ensuring a majority of the services agree on the results.

**Deployment:** This technique would be integrated into the code of programs that wanted to use these new protocols.

**Execution Overhead:**

- Additional resources required to run many of the same services
- Additional time added while the protocols are reaching agreement

**Memory Overhead:**

- Additional resources required to run many of the same services

**Network Overhead:**

- IPSec adds an additional 24 bytes to each packet header
- Additional network traffic by all the broadcasting and exchanging of messages while the protocols are reaching agreement
- IPSec adds an average 30% latency for each protocol
- Can have an impact on network throughput for large volumes of traffic

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** There are limited applications that can benefit from a protocol like RITAS. Additional protection techniques are certainly necessary.

**Weaknesses:** One large weakness of this technique is that is can only tolerate $f = \lfloor \frac{(n-1)}{3} \rfloor$ compromised processes. More importantly, RITAS does not provide any protection against one-node compromises. Attacks like data leakage (exfiltration) are still a concern.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique is only useful against integrity attacks. If an attacker were trying to manipulate the output of programs or the data being passed around on the network, this

would increase his workload because the attacker would need to compromise a certain percentage of processes as opposed to just one. This also adds additional impact to the attacker if he were trying to take down the process or service. Running multiple copies provides overall greater resiliency if one or more were to fail.

**Availability:** This technique was prototyped by the author but the code was not publicly released.

**Additional Considerations:** This method is very limited in scope in that it deals with a particular problem (message passing) with a protocol. Much of this work is very theoretical.

- May not work well for applications that require very low latency or streaming
- Needs to work with applications where multiple instances would produce the same output
- Running potentially numerous instances of a process will likely increase the maintenance workload and overhead

**Proposed Research:** This technique abstracts out what the processes are actually doing or how they are setup. Adding randomization or diversity techniques to the individual processes or machines they reside on would further increase the workload of the attacker assuming that such diversity did not result in the processes producing different outputs. If all processes were running on similar systems, if an attacker was able to compromise one, they may also be able to compromise many with similar methods degrading the effectiveness of this technique.

**Funding:** European Network of Excellence, FCT (Portugal)

### 6.4 NASR

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource and Scanning

**Details:** This technique [133] was designed to mitigate and slow the effects of an IP address hitlist-based worm. It does not actually stop any specific attacks. It can be used on some level to reduce the effectiveness of scanning attacks. The information collected from these attacks would change as the internet protocol addresses of the systems changed.

**Description:**

**Details:** Network Address Space Randomization (NASR) is a technique that involves changing the IP address of systems more frequently. The authors modified a Dynamic Host Configuration Protocol (DHCP) server to have short IP address leases and to force an IP address change when a lease expires. The side effect of changing these IP addresses constantly is that persistent or active connections would be dropped during the address change. The authors developed sensors to attempt to profile the services on a system and the connections on a system. If a system has many connections that would be dropped, the changing of the address is delayed. There is a hard limit where a system will be forced to change its IP address as well if it has not changed for a long time. There are some types of systems that have constant persistent connections that would be excluded from this technique. There are also some systems that require a static IP address that would be excluded as well. Domain Name System (DNS) servers can be used for outside access to servers and services to mitigate the impact a constantly changing IP address would have on end users.

**Entities Protected:** This technique helps mask the identify of systems and servers from information collection and targeted attacks.

**Deployment:** This technique would generally be implemented in segments of a local area network (LAN).

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- Dropped connections due to IP address changes during interactions

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** This technique only slows down certain types of attacks but relies on other detection mechanisms to detect these attacks.

**Weaknesses:** This technique does not protect systems that rely on static IP addresses or systems that use DNS. If a system is using DNS, the attacker can just point to that address and does not have to worry about the actual IP address. The effectiveness of this technique is also reduced if there is not a large enough pool of IP addresses available.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☒ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** This may impose some overhead on the attacker to maintain a mapping of systems but it, by itself, does not stop an attacker from launching any attacks against a system or server.

**Availability:** This technique was prototyped by the authors but code was not publicly released.

**Additional Considerations:** This technique does not provide any protection against targeted attacks or attacks that can reach the machine using higher level protocols. It also does not protect against client-side attacks (*e.g.*, browsing to a malicious website). The technique is very limited in scope and can break many functionalities.

**Proposed Research:** This technique could be extended to have a larger pool of addresses to use for randomization. Another idea would be to extend it to randomize network properties such as port numbers. An external abstraction layer or proxy could be used to translate addresses coming into this network such as a Network Address Translation (NAT) device. This would make the individual internal systems transparent to the outside world. Combining this technique with other network technologies that manage connections between systems could reduce the amount of dropped connections due to an address change.

**Funding:** European Commission/Information Society Technologies, Greek Secretariat for Research and Technology

### 6.5 MUTE

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource and Scanning

**Details:** In this technique [134], the shifting IP addresses would make it more difficult for an attacker launching denial of service type attacks against individual systems in the network. The shifting IP addresses, port numbers, and packet routes would also make it more difficult for an attacker running scans on the network trying to identify what systems are there as well as the services running on those systems.

**Description:**

**Details:** A Mutable Network (MUTE) is a technique that involves changing Internet Protocol (IP) addresses, port numbers, and routes to destinations inside of a network. This technique is proposed to be implemented as a sort of virtual overlay to the existing network so the original IP address and information on the systems never changes. All traffic is routed independently over this virtual overlay. Synchronization of IP address information across the network would be done across encrypted channels. There would also be mechanisms in place to apply transformations on the network traffic to confuse the tools attackers are using to identify the services and hosts. The packets can be changed based on rules distributed amongst routing entities. It can change the source and destination IP address as well as source and destination ports. There is a sense of possible network configurations so packets can be rerouted to get to their destination via a different path. There would also be policies in place to ensure any global network requirements are satisfied.

**Entities Protected:** This technique helps mask the identities of systems inside of a network. By changing the information associated with systems, information collected by attackers would be constantly shifting.

**Deployment:** This would be deployed on all devices capable of routing network traffic and wish to participate in this technique.

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- There may be unknown, but significant overload of network infrastructure including routers and switches

- The extra routing overhead may break the network infrastructure

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** This technique can be combined with other network-based detection and monitoring systems.

**Weaknesses:** One potential weakness of this technique would be if an attacker could still attack the original IP address of the machine since it does not change. Another possible weakness is if the IP address information does not change fast enough. An attacker could do enough reconnaissance to figure out what they need then launch their attack before the change happens. In addition, if any systems are using a Domain Name System (DNS) address, this will be updated with the IP addresses and an attacker could target a machine via that address. More importantly, this technique only protects reachability. It does not provide any protection against client-side attacks (*e.g.*, browsing to a malicious website).

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☒ Limit Movement

☒ Disable Movement

**Impact on Attackers:** This technique could have varying levels of impact on an attacker depending on what the attacker is trying to accomplish. If an attacker is trying to disrupt the network, it could make flooding attacks more difficult if they did not have access to DNS addresses. Since this technique is not supposed to disrupt active connections, if an attacker can collect the information they need before a switch and establish a connection to the system, they may not be as impacted as much by this technique.

**Availability:** This is a research idea and was not implemented.

**Additional Considerations:** A technique like this could have impact on applications or services that require constant connections or could disrupt current connections. More importantly, the protection offered is very limited. It does not protect against client-side attacks. The technique can also have severe scalability issues.

**Proposed Research:** Since this is a proposed idea, many aspects are still undefined. It is not clear how the technique could be put in place such that it would not affect active connections or running services. It is also not clear how to handle adding or remove systems from this network or how far it would scale. It must also come up with a way to be fast enough to impact attackers while not overloading the systems or network with the changes. Finally, how this needs to be implemented into a system would need to be investigated as well. It is not clear how the underlying actual network is protected if at all. If an attacker is still able to get in through the original network that does not change then it defeats the purpose of this technique.

**Funding:** Unknown

### 6.6 DYNABONE

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource

**Details:** This technique [135] is designed to mitigate a specific type of resource attack known as denial of service. It does this by dynamically rerouting network traffic away from virtual overlay networks that are being flooded.

**Description:**

**Details:** Dynamic Backbone (DynaBone) is a technique that involves creating multiple inner virtual overlay networks inside of a larger outer virtual overlay network. Each of the inner networks can be using a different networking and routing protocol or hosting a different service to increase diversity amongst them. Each host in the outer overlay network is not aware of the inner networks giving the appearance of only one network. The entry points to these internal overlays have a collection of sensors that monitor performance and possible attack traffic. Based on the conditions of the networks, it decides which internal network to use. If an internal overlay is detected to be under attack or is suffering performance issues, traffic can be routed through different overlays (dynamic network aspect of DynaBone). This technique is built on top of X-Bone that is a dynamic network overlay technique that allows multiple simultaneous virtual overlays to coexist. It allows network topologies to be dynamically created and used by applications. Hosts and networking devices can participate in multiple overlays. This can also be set up so various physical paths in the network are unique to different overlays.

**Entities Protected:** This technique aims to protect the availability of services on a network. Traffic can be dynamically rerouted or routed through multiple paths simultaneously.

**Deployment:** This would be deployed on all entities participating in the virtual network.

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- Depending on the various networking and routing protocols that are being deployed with this technique, they can add additional latency and reduce bandwidth. These can include encryption and authentication protocols/algorithms.

- The impact of additional routing and load on the network infrastructure is unknown.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** A good detection mechanism to detect when an overlay is under attack. This technique assumes that attacks can be detected.

**Weaknesses:** The inner overlays may not be sufficiently disjoint and it could be the case that the loss of certain hosts/networking devices/routes can severely affect the overall network. If the service is not distributed, it is also possible for an attacker to take the service out by flooding the service provider. Also this technique does not provide any protection against targeted attacks or data leakage (exfiltration) attacks. This technique does not provide any protection against client-side attacks.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☒ Disable Movement

**Impact on Attackers:** This technique has varying levels of impact on an attacker depending on the goals of the attacker. If an attacker is trying to take down a service by flooding hosts or network infrastructure, this technique could make it more difficult for him. If an attacker were attempting to take out a service via other means such as attacking the service directly, this technique would be less effective.

**Availability:** This technique was prototyped by the authors but the code was not publicly released.

**Additional Considerations:** This technique is limited in the protection it provides. It does not provide any protection after a host is reached. More importantly, it does not protect against client-side attacks. In addition, this technique can severely impact functionality by limiting communication.

**Proposed Research:** One idea for this technique is go combine it with techniques that also increase the resiliency of the end service as well. This could include techniques that run multiple instances of a service. This would increase the overall availability of the service by making it more difficult for the attack to disrupt the network and the end service.

**Funding:** DARPA, Air Force Research Laboratory

### 6.7 ARCSYNE

**Defense Category:** Dynamic Networkss

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Scanning and Resource

**Details:** This technique [136] helps mitigate scanning-related attacks by continually changing Internet Protocol (IP) addresses. Hopping makes the life of the collected information limited. This technique would also help mitigate some resource attacks related to denial of service (DoS) attacks. It is presumed that the gateways do not have a global Domain Name System (DNS) address so an attacker would need to target a large set of IP addresses simultaneously or constantly change the target for a DoS attack to reach the target.

**Description:**

**Details:** Active Repositioning in Cyberspace for Synchronized Evasion (ARCSYNE) is an IP address hopping technique implemented at Virtual Private Network (VPN) gateways. The functionality is implemented into the kernel of the gateway operating system. Each gateway participating in the hopping shares a secret and a clocking mechanism. At each clock tick, the gateways compute a new IP address based on the secret and the clock. Each gateway also computes what the other gateway IP addresses will become. The IP hopping does not disrupt connections between gateways including streaming services. In order to account for packets that are delivered shortly after an IP address change, the gateways can still accept those packets up to a grace period. This grace period should be approximately equal to the time it takes for one packet to go from one gateway to another. This technique has been tested with a large number of standard network protocols and services.

**Entities Protected:** This technique aims to protect the discoverability and reachability of the VPN gateways between networks. The presumptions is that if an attacker cannot locate and reach a gateway before the IP hopping takes place, they will not be able to launch an effective attack against that gateway or the systems behind that gateway.

**Deployment:** This technique would be deployed on the VPN gateways in a network. Clients that are operating within this private network should not need to be modified.

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- Changing the address information in packets may have an impact on the delivery times.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** The method for deriving and delivering the shared secret is secure.

**Weaknesses:** One weakness of this technique would be having an insufficiently large pool of IP addresses to use for hopping. If the pool is too small, an attacker could focus more on the limited addresses or be able to predict which addresses will be next with better accuracy. In addition, this would give an attacker with adequate resources the ability to launch DoS type attacks against the entire limited address space cutting off communication at that gateway. If it is possible for an address to be chosen twice or more in a row due to the random selection, it might give an attacker larger windows to mount an attack. If the systems that are part of the VPN are also part of a local network, it may be possible for an attacker to compromise a system within that local network and then launch an attack on the systems that are part of the VPN directly. This technique is also not effective if an attacker is able to locate the target and mount an attack before the hopping takes place. If an attacker can analyze traffic, he or she may be able to use other aspects of the network traffic besides the address to determine where the current targets are located. In fact, the protection offered by this technique is only based on limited reachability. For example, this technique provides no protection against client-side attacks (*e.g.*, browsing to a malicious website).

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☒ Limit Movement

☒ Disable Movement

**Impact on Attackers:** This technique increases the amount of work an attacker has to do to discover the targets if he is using IP scans. An attacker would need to scan random IP addresses in order to discover target that is constantly changing addresses as opposed to scanning for a fixed host then mounting an attack. However, this technique does not provide any protection against an adversary that can reach a host using higher-level protocol information (web browsing or application-level communication).

**Availability:** This technique is being prototyped and tested by the Air Force Research Laboratory as a proof-of-concept but does not appear to be publicly available at this time.

A similar commercial product is available from Invicta: `http://www.invictanetworks.net`

Another similar commercial product is available from Telecordia: `http://www.telcordia.com`

Another commercial product that combines a similar IP-hopping technique with multi-factor authentication and role-based access control is available from Cryptonite: `http://www.cryptonitenxt.com` [137]

**Additional Considerations:** The protection only focuses on masking IP addresses. Note that a host can be reached by many other means: browsing to a website, application-level communication, P2P traffic, etc. The limitations imposed by the technique and the functionality impacts may outweigh the protection offered.

**Proposed Research:** This technique might offer more protection if it was implemented on individual systems as opposed to at the gateways. This would help protect against any scanning or attacks that are targeting the participants in the VPNs directly. Additional randomization of protocol fields or the inclusion of dummy traffic might also offer more protection for attackers that are able to perform traffic analysis. However, implementing at the level of individual hosts significantly increase the overhead.

**Funding:** Air Force Research Laboratory

## 6.8 RANDOM HOST MUTATION

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Scanning

**Details:** This technique is designed to make a host untraceable by network scanning in order to mitigate reconnaissance attacks that reveal the IP addresses of a host. It does not stop any specific attacks. While the technique is in use, the host address information gained by an attacker performing network scanning would change over time and have limited value as the technique changes the routable IP addresses of hosts.

**Description:**

**Details:** Random Host Mutation (RHM) [138] is a technique that involves changing the routable IP addresses of systems frequently. RHM keeps the real IP addresses (rIP) of end hosts unchanged and creates for these hosts routable virtual IP addresses (vIP) that are short-lived and changed randomly, consistently, and synchronously in the network. Only at the network edges (subnet) close to the destination is a vIP address translated into the rIP address by a special gateway (MTG). Hosts with changing vIP addresses are reachable by hostnames that are translated by the Domain Name System (DNS) service into rIP addresses, then the MTG translates these to vIP addresses before providing them to the source hosts. Alternately, one host can access another using its rIP address; in that case, the MTG requests authorization from a controller (MTC) and provides the routable vIP of the destination host if access is allowed. During address mutation, sessions are maintained because RHM allocates and reserves new addresses for hosts until their existing flows are terminated.

Adaptations of the technique include using OpenFlow virtual switches [139] or TAP virtual network kernel devices [140] to provide address mapping.

**Entities Protected:** This technique helps mask the identity of end hosts from information collection and targeted attacks.

**Deployment:** This technique would generally be implemented in segments of a local area network (LAN).

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- Address-space overhead caused by assigning endpoints multiple vIPs in order to maintain sessions during mutation. The faster the mutation rate, the higher the overhead.
- Routing-update overhead is proportional to the routing-table size after each coarse-grained mutation interval.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** This technique can be combined with other network-based detection and monitoring systems.

**Weaknesses:** It is possible there may be not enough IP address space to support the desired level of movement. Configuring the RHM system may be difficult, since hosts have configurable security levels that affect their mutation rates, and misconfiguring these could lead to high address space overhead or routing overhead. Another possible weakness is if the vIP address information does not change fast enough, perhaps due to misconfiguration, allowing an attacker to do reconnaissance and launch an attack before the change happens. In addition, the technique provides no protection if an attacker has the DNS name for their target and can therefore get the current vIP address of the target from the MTG. This technique does not provide any protection against client-side attacks (*e.g.*, browsing a malicious website).

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☒ Limit Movement

☒ Disable Movement

**Impact on Attackers:** This technique could have varying levels of impact on an attacker depending on what the attacker is trying to accomplish. If an attacker is trying to disrupt the network, it could make flooding attacks more difficult if they do not have access to DNS addresses.

**Availability:** The authors have prototyped the system and evaluated it on a small university network and with simulation. It is not publicly released.

**Additional Considerations:** This technique could have scalability issues related to address-space overhead when the security requirements for hosts are high.

**Proposed Research:** This technique could be deployed in concert with other moving target techniques; in [138], the authors evaluate it in isolation, and only focus on evaluating the overhead and the effectiveness of slowing down routing-worm propagation. The reliability and security of the RHM architecture itself were not studied.

**Funding:** Unknown

## 6.9 OPENFLOW RANDOM HOST MUTATION

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Scanning

**Details:** This technique [139], which is related to [138], was designed to make hosts in an OpenFlow software-defined network (SDN) untraceable by network scanning in order to mitigate reconnaissance attacks that reveal the IP addresses of hosts. It does not stop any specific attacks. While the technique is in use, the host address information gained by an attacker performing network scanning would change over time and have limited value as the technique changes the routable IP addresses of hosts.

**Description:**

**Details:** OpenFlow Random Host Mutation (OF-RHM) is a technique that involves changing the routable Internet Protocol (IP) addresses of systems frequently. It is a variation of RHM [138], which keeps the real IP addresses (rIP) of end hosts unchanged and creates for these hosts routable virtual IP addresses (vIP) that are short-lived and changed randomly, consistently, and synchronously in the network. Essentially, OF-RHM uses OpenFlow switches (OF-switches) as the RHM gateways and the OpenFlow controller (OF-controller) as the RHM controller. In OF-RHM, the OF-switch translates vIP addresses to rIP addresses as specified by an OF-controller. The other tasks of the OF-controller in OF-RHM include 1) coordinating vIP mutations across switches using OpenFlow messages, 2) handling DNS updates, 3) choosing the new address space for end host vIP mutations, and 4) installing flow rules and actions in the OF-switches that support the RHM behavior.

New flows to DNS-named hosts reach the controller, which modifies the DNS response to use the vIP for the destination and sets the DNS TTL to be very small. When the flow is initiated, the OF-switch sends the controller the initial packet, and the controller installs flow rules in the switches in front of the source and destination OF-switches. These rules translate between the rIP address and vIP address at the endpoints; all other switches in the path just route the flow based on vIP addresses.

As in RHM, authorized users can also reach a host using its rIP address if they have it. In this case, when the new flow starts, the controller gets the initial packet and must decide whether the user is authorized to access the host via its rIP. If authorized, the controller installs rIP-vIP translation rules along the route. During address mutation, flows are maintained because rules will not be evicted from the OF-switch while they are in use. Flow rules with old addresses eventually expire and are evicted from OF-switches. After a host's vIP address mutates, the controller begins installing flow rules with the new address as it gets new packet-in messages for the flow from the OF-switches.

**Entities Protected:** This technique helps mask the identity of end hosts from information collection and targeted attacks.

**Deployment:** This technique would generally be implemented in segments of a local area network (LAN) with OF-switches at the edges.

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- Address-space overhead caused by assigning endpoints multiple vIPs while flows are maintained. The faster the mutation rate, the higher the overhead.
- Flow-table size overhead (number of rules) on a OF-switch is related to the address-mutation rate, the number of hosts on the switch, and the rate at which flows terminate.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** This technique depends on having OpenFlow-enabled switches in the network in front of the hosts that will be protected.

**Weaknesses:** It is possible there may not be enough IP address space to support the desired level of movement. Configuring the OF-RHM controller may be difficult, since individual end hosts can have configurable security levels that affect their mutation rates, and misconfiguring these could lead to high address space overhead or flow-table size overhead. Another possible weakness is if the vIP address information does not change fast enough, perhaps due to misconfiguration or very lengthy flows that keep rules alive on switches longer than expected. In this case, an attacker could do reconnaissance and launch an attack before the change happens. In addition, the technique provides no protection if an attacker has the DNS name for their target and can therefore get valid flows to the destination installed in the network. This technique does not provide any protection against client-side attacks (*e.g.*, browsing a malicious website).

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☒ Limit Movement

☒ Disable Movement

**Impact on Attackers:** This technique could have varying levels of impact on an attacker depending on what the attacker is trying to accomplish. If an attacker is trying to disrupt the network, it could make flooding attacks more difficult if they do not have access to DNS addresses.

**Availability:** The authors have prototyped the system and evaluated it a Mininet simulation. The implementation is not publicly released.

**Additional Considerations:** This technique could have scalability issues related to address space overhead and flow table size when the security requirements for hosts are high.

**Proposed Research:** This technique could be evaluated and deployed in concert with other moving target techniques. It is not clear whether the technique could be seamlessly deployed into existing SDN networks that are running other controller applications without interfering with those, or whether the flow-table size overhead would significantly affect the flow initiation speed. The evaluation performed in [139] focuses on the effectiveness of OF-RHM against external scanning and target discovery by worms, but evaluating the defense's protection against other attack models like distributed denial of service (DDoS) would be helpful, as the authors note.

**Funding:** Unknown

### 6.10 SPATIO-TEMPORAL ADDRESS MUTATION

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Scanning

**Details:** This technique is designed to mitigate reconnaissance attacks that reveal the IP addresses of hosts by scanning. It does not stop any specific attacks. While the technique is in use, the host-IP binding information of any given destination host changes in two ways: it varies depending on the source host ("spatial") and the time ("temporal"). Therefore, even attackers who can quickly perform scanning cannot exploit reaching those addresses from any source host.

**Description:**

**Details:** This technique [141] works by varying the host-IP bindings dynamically based on the location of other hosts trying to use any given IP and the time at which the IP is used. The goal is to add an extra dimension of dynamicity in host-IP bindings compared to other time-dynamics-only IP-hopping techniques, like Random Host Mutation (RHM) [138]. Using the technique, the real IPs do not change, and the host platforms require no changes. Instead, at any point in time, each host maintains a set of ephemeral IPs (eIPs) by which they can address each other host. The eIP that host $A$ uses to reach host $B$ is valid only during a changing time interval, and other hosts cannot use that eIP to reach $B$ even when it is valid for $A$.

Like RHM, spatio-temporal mutation uses a controller to compute new mappings and it uses gateways that translate between real IPs and eIPs. The controller has the full view of the network. The gateway of the subnet where DNS is located is in charge of modifying each DNS reply to replace the real IP with the chosen eIP, and replace the DNS TTL with the chosen randomization interval length.

Two strategies are used to determine the appropriate eIP bindings:

1. Random mutation, which involves selecting an address from the unused address space by sampling a uniform distribution, and
2. Deceptive mutation, which constructs bindings to deceive the adversary, *e.g.*, to assign *potentially unattractive* eIPs to potentially attactive or vulnerable hosts. Intuitively, an eIP might be unattractive to a host $H$ if it is possible the host at that eIP has already been reached from $H$. The technique can compute this using the real network topology.

The deceptive mutation strategy costs more in overhead compared to random mutation, but the authors argue that it is more difficult for persistent attackers to overcome it. Therefore, the technique uses deceptive mutation when it detects signs of a scanning attack, and random mutation otherwise. The detection indicator is an abnormal increase in the size of connected components in the network communication graph during a time interval. In other words, if hosts are communi-

cating with each other more than normal during a period of time, that could be a sign of an attack like worm propagation, so the more aggressive mutation scheme is used.

**Entities Protected:** This technique helps mask the identity of end hosts from information collection and targeted attacks.

**Deployment:** This technique would generally be implemented in segments of a local area network (LAN). It could be deployed in a software-defined network (SDN), where the mutation controller doubles as the OpenFlow controller and the gateways are OpenFlow switches, similar to OpenFlow Random Host Mutation [139].

**Execution Overhead:**

- Execution overhead on the controller for computing the deceptive or random mutations after each interval. See [141] for the full time-complexity analysis. Empirical overhead measurements are not presented.

**Memory Overhead:**

- None

**Network Overhead:**

- Address space overhead caused by assigning endpoints to multiple eIPs in order to maintain sessions during mutation. The faster the mutation rate, the higher the overhead.
- DNS traffic overhead related to spatial mutation. Temporal mutation forces all clients to renew eIP mappings by querying the authoritative DNS service at shorter than usual intervals.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☒ Custom Programmer (General Knowledge)
- ☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

- ☒ Seamless
- ☐ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

**Kill-Chain Phases:**

- ☒ Reconnaissance
- ☒ Access
- ☐ Exploit Development
- ☐ Attack Launch
- ☐ Persistence

**Interdependencies:** This technique can potentially be combined with other network and endpoint monitoring tools. It assumes that the control plane (*e.g.*, controller, DNS) is not the target of attack.

**Weaknesses:** There is some chance that an attacker can overcome the temporal mutation and spatial mutation by quickly performing reconnaissance and launching an attack from the same host. Like RHM, it is also possible there may be not enough IP address space to support the desired level of movement. Attacks on the DNS service or controller could disable the movement by slowing down or stopping the eIP binding refresh. This technique does not provide any protection against client-side attacks (*e.g.*, browsing a malicious website).

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement

☒ Limit Movement

☒ Disable Movement

**Impact on Attackers:** Reconnaissance attacks will generally be slowed down by this technique, and the information gained might have limited or no value depending on when it is used and from which foothold host. A persistent adversary might try scanning frequently and from each host she is able to target and reach, but that increases her chance of being detected (and increases the likelihood that the technique selects the deceptive mutation strategy, which is more difficult to overcome). Spatial mutation mitigates honeynet mapping attacks, because an eIP can appear either used or unused depending on the source host.

**Availability:** The authors prototyped the technique using an OpenFlow network and evaluated it using Mininet. The implementation is not publicly available.

**Additional Considerations:** None

**Proposed Research:** The technique has been evaluated in a mostly theoretical way, *e.g.*, deception and detectability metrics defined by the authors are evaluated as other configurable variables, like address-space size, change. A next step is performing an end-to-end study of the overhead and security impact of a realistic deployment. In particular, evaluating whether the spatial mutation aspect of eIPs is worth the added overhead, compared to only temporal mutation, would be interesting (*i.e.*, head-to-head comparison with RHM).

**Funding:** NSF

### 6.11 AVANT-GUARD

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource

**Details:** This technique includes a function that can restrict flows based on events that are triggered by traffic dynamics. While the technique is in use, attacks like denial of service (DoS) that flood a targeted server could be interrupted.

**Description:**

**Details:** AVANT-GUARD [142] is a software-defined networking (SDN) technique that includes two security functions: 1) *connection migration*, which defends against control-plane saturation attacks, and 2) *actuating triggers*, which dynamically activates flow rules on the the data plane in response to traffic dynamics that might indicate an attack or disrupt targets on the network. The latter, actuating triggers, may be considered a moving target defense because flows can be affected by rules that are dynamically activated under conditions that are unknown to the attacker.

Actuating triggers in AVANT-GUARD are implemented using new extensions to the OpenFlow protocol and using a modified data plane. The control plane defines conditions and registers them with rules in the data plane. Whenever the data plane receives a packet, it runs a trigger-evaluation function added by AVANT-GUARD. Three types of conditions are supported: 1) payload-based, which uses a 1-bit condition (flag) that indicates whether to send payloads from the matched flow to the control plane; 2) traffic-rate-based, which uses a 22-bit condition that specifies the traffic rate (*i.e.*, packets per second [PPS], bits per second [BPS], or raw count), a comparator, and a value to compare the current rate against, and reports the network status to the control plane when triggered; and 3) rule-activation, which is similar to the traffic-rate-based type but activates a rule stored on the data plane when triggered (*e.g.*, to block a flow with heavy traffic without having to transact with the control plane).

**Entities Protected:** This technique helps protect endpoints in the network from DoS or other resource attacks by using traffic-rate triggers. It can also be used to report traffic status or payloads from targeted flows for use in other control plane security tools, *e.g.*, ones that detect distributed denial of service (DDoS) attacks or malicious payloads.

**Deployment:** This technique can be deployed to SDNs using OpenFlow software switches, or implemented with modifications to hardware OpenFlow switches.

**Execution Overhead:**

- Evaluating conditions adds minor execution overhead on the data plane. The payload-based condition checks one bit and has virtually no overhead. In [142], the authors found that the traffic-reporting condition added 0.322 $\mu$s to a flow and rule activation added 1.697 $\mu$s in their experiments.

**Memory Overhead:**

- The technique needs additional TCAM or SRAM storage on the data plane for stored rules that get activated by triggers.

**Network Overhead:**

- If conditions on the data plane trigger communications with the control plane—through payload delivery or trigger reporting—there may be network overhead.

**Hardware Cost:**

- SDN control and data planes, if not already installed.

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☒ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☐ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

**Interdependencies:** This technique is dependent on having an SDN infrastructure. The control plane is assumed to be able to define and register conditions using the extended OpenFlow protocol. The network-status reporting trigger could be used by other tools running on the control plane as a timely alternative to those tools that poll the data plane for flow statistics.

**Weaknesses:** One weakness of this technique is that the conditions supported for dynamic rule activation are limited to rates that can be computed efficiently on the data plane (*e.g.*, PPS, BPS). There is also the need for additional switch memory in the data plane (TCAM or SRAM), which could be expensive. While it may be possible to share the existing TCAM or SRAM in an OpenFlow switch, this might limit the number of flow rules that could be installed, especially if the control plane is pushing rules using applications in addition to AVANT-GUARD. It is possible that could limit the ability of flow rules to change dynamically. It is also possible that an attacker could infer some conditions for the rule-activation triggers by monitoring his own traffic statistics, predict when rules will activate, then operate in a way that does not trigger the rule activation. In this case, some attacks might succeed if conditions that block flows are not aggressive enough.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☒ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique makes it difficult for attackers to perform some attacks, like DoS attacks, that result in abnormal flow-traffic statistics. As noted above, an attacker may be able to succeed in a DoS attack depending on factors like the trigger thresholds, behavior of the target service, and whether some reconnaissance is possible. This technique alone does not prevent an attacker from sending a malicious payload in a flow that stays below the traffic-rate trigger conditions, though its payload trigger could be used to complement other defenses on the control plane that perform packet inspection.

**Availability:** The authors have prototyped the system using a software OpenFlow data plane. The prototype is not publicly released.

**Additional Considerations:** The technique requires extending the OpenFlow protocol (*e.g.*, adding a "trigger" message sent from the data plane to the control plane), and modifying both the control plane and the data plane. The protection does not include network access control, only statistics-based rule activation and signaling to the control plane. Scalability may be an issue for the flow-table memory. Also, the control plane must handle the security/usability tradeoff when using rule-activation triggers; aggressive conditions can result in false-positive activations and weaker ones might allow attacks to succeed. It is possible that legitimate flows could trigger rules that interrupt these flows.

**Proposed Research:** Combining this with other network technologies or control-plane analytics could improve the protection it offers. For example, traffic-rate conditions could be determined by the control plane based on automated testing of the services in the network and their ability to handle traffic. In addition, the technique has not been evaluated empirically as a hardware-based implementation, which would provide insight about the feasibility of the method.

**Funding:** DARPA, United States Air Force

### 6.12 IDENTITY VIRTUALIZATION FOR MANETS

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Scanning, Spoofing, Resource

**Details:** This technique is aimed at re-randomizing virtual node identities in a mobile ad-hoc network (MANET) and updating these identities in the network in a decentralized way. This can make scanning difficult and prevent spoofing of messages, since legitimate nodes authenticate senders cryptographically.

**Description:**

**Details:** This technique [143] protects MANETs against scanning and attacks that target nodes using their identities (*e.g.*, real IP addresses in a traditional network). Each node maintains a virtual ID at any point in time that is associated with its real ID. Only virtual IDs are used for addressing in legitimate communications; they are translated into real IDs using a translation service with which the network layer is augmented. In order to avoid attackers scanning and utilizing the virtual IDs, nodes change their virtual IDs at a node-specific validity interval, then send updates that other legitimate nodes can verify using a special update protocol.

Generating virtual IDs and translating them is done in a decentralized network using cryptographic hashes. It is assumed that legitimate nodes have two shared secrets that are used 1) to encrypt join/leave request packets so that they cannot be spoofed by illegitimate nodes, and 2) to generate virtual IDs via hash chains that other legitimate nodes can authenticate. When a node changes its virtual ID by selecting the next hash in its chain, it broadcasts its new virtual ID alongside with the index of that ID in its own hash chain. Other legitimate nodes in the network can use this, with the shared secret and hash function, to determine which real ID the update corresponds to before updating their own translation tables with this mapping.

**Entities Protected:** This technique helps protect nodes from scanning and spoofing attacks from external nodes, and protects the network from routing attacks like flooding and blackholes.

**Deployment:** This technique is deployed on nodes in a MANET.

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- Updating a single node's virtual ID across the MANET causes overhead at each other node.
- Packet loss increases with the frequency of node ID updates.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** The technique assumes that shared secrets are securely distributed to legitimate nodes.

**Weaknesses:** A major weakness is that network performance (*e.g.*, well-formed packets delivered as expected between legitimate nodes) degrades badly as the network scales in the number nodes or when the frequency at which nodes change virtual IDs increases. Mitigating this means weakening the security by reducing the frequency, but if the frequency is too low, attackers could spoof messages or perform attacks like route invalidation using overheard IDs that temporarily remain valid. This technique provides no protection against attacks by insiders, who could send malicious payloads, spoof messages from other legitimate nodes, or perform routing or join-/leave-request forgery attacks.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☐ Limit Movement

☐ Disable Movement

**Impact on Attackers:** Attackers may be able to learn the virtual IDs of nodes while they are still valid and use them for some attacks. The technique makes this more difficult by providing protocols that let nodes change IDs over time and securely update others about the change. The technique does not protect against side-channel attacks, malicious insiders, or brute-force attacks against the cryptographic mechanisms.

**Availability:** The authors prototyped the defense in the NS-2 simulator. It is not publicly released.

**Additional Considerations:** The technique requires configuration of the validity interval that nodes use to mutate their virtual IDs. Performance in the network downgrades rapidly if the update frequency is too high. On the other hand, the protection provided by the technique is weaker when the frequency is lower. For the threat model considered, nodes that join the network are assumed to be legitimate if and only if they hold the shared secrets required by the network, which must be distributed securely.

**Proposed Research:** More research could be done to evaluate the technique beyond its effect on network performance. For example, it would be helpful to know how security measures improve or decrease in response to changing the update frequency on nodes or the total number of nodes in the network.

**Funding:** Unknown

## 6.13 MORPHING COMMUNICATIONS OF CYBER-PHYSICAL SYSTEMS

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Scanning

**Details:** This technique is aimed at preventing traffic analysis attacks on network sessions of a cyber-physical system (CPS). It provides an algorithm to distribute packets over multiple sessions such that the sessions appear to have target traffic-statistic distributions that are indistinguishable from background internet traffic, while meeting deadline requirements for packets.

**Description:**

**Details:** This technique [144] protects a CPS from having its network sessions identified by an attacker via a traffic analysis attack. This is useful because CPSs are widely deployed and a significant portion of their traffic may be relayed through existing shared infrastructure. While CPS traffic is typically encrypted, it may still be vulnerable to traffic analysis that lets an attacker eavesdrop and identify sessions based on timing side channels like the inter-packet delay (IPD), which may be characteristic of the system (*e.g.*, a controller and sensor or actuator communicating at intervals). Other traffic obfuscation algorithms exist, but current approaches are likely to violate the delivery-time requirements that CPSs often have, so this technique directly incorporates these requirements into the algorithm.

The technique provides an algorithm called CPSMorph that distributes packets into sessions in order to best match the IPD distributions of random sessions in the background while meeting sending-deadline requirements for packets. Redundant packets are delivered to help form the desired distributions, adding overhead to the network traffic. The algorithm tries to minimize this overhead while respecting the constraints on message delivery time and each session's traffic profile converging to its target distribution. Its authors consider CPSMorph a moving-target defense because it makes active sessions belonging to the CPS "moving targets" that nondeterministically change their traffic profiles and cannot easily be classified as distinct from other sessions in the background.

**Entities Protected:** This technique helps protect network sessions from being identified as belonging to the CPS via IPD-based traffic analysis attacks.

**Deployment:** This technique is intended for deployment at agents in a CPS that run the algorithm and communicate across shared infrastructure.

**Execution Overhead:**

- CPS agents executing the `send` routine. Execution overhead is not reported.

**Memory Overhead:**

- CPS agents may create multiple active sessions to distribute the messages.

**Network Overhead:**

- Redundant packets are used to disguise session statistics. The technique's authors found on average 51.84% of packets sent were redundant in their experiments.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

Source code on the CPS agents may need to be modified to implement CPSMorph, *e.g.*, to expose message deadlines to the algorithm.

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** The technique assumes that the network delivery delay can be measured accurately and used as an argument in the algorithm.

**Weaknesses:** The defense provides no protection outside its threat model of traffic-analysis attacks. It only focuses on first-order distributions of IPDs, but other analyses of the traffic might identify the CPS, which could let an attacker overcome the movement in CPSMorph. If the CPS allows for little or no delay when a message is sent, then the overhead gets worse because more active sessions may be created to distribute the packets. Furthermore, the technique does not defend against malicious data sent to the CPS, or protect CPS components from other attacks (*e.g.*, physical attacks on sensors, active attacks on the relay network). If an attacker can compromise communications in one CPS agent using one of these other attacks, they could disable movement or change the algorithm so that session IPD distributions are identifiable by traffic analysis.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☒ Limit Movement

☒ Disable Movement

**Impact on Attackers:** The technique makes it more difficult for an attacker to identify CPS sessions using IPD-distribution analysis on eavesdropped traffic. This type of attack can already fail without CPSMorph depending on the characteristics of the CPS, so the impact of the defense is limited.

**Availability:** The authors implemented the algorithm and evaluated it using simulation. Pseudocode is given in [144]. The implementation is not publicly available.

**Additional Considerations:** The technique has very limited protection at the cost of significant network overhead. CPSMorph may not be suitable for CPSs with constrained resources or with strict performance requirements (*i.e.*, little flexibility in how long messages can be delayed).

**Proposed Research:** More research could be done to evaluate the technique under real conditions. It is not clear how the number of active sessions that CPSMorph creates scales with message-deadline requirements that are typical for real CPSs. It would also be interesting to evaluate

the technique factoring in uncertainty around the expected network-delivery delay, *e.g.*, using a distribution instead of a single estimate in the algorithm.

**Funding:** Unknown

### 6.14 CLOUD-ENABLED DDOS DEFENSE

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource, Scanning

**Details:** This technique [145] defends against distributed denial-of-service (DDoS) attacks, particularly aimed at Internet services. It uses selective server replication in an elastic cloud to shuffle clients from attacked servers into newly instantiated server replicas that have new network locations, known only to the clients migrated to them. The attack model assumes a botnet and two types of bots participating in the DDoS attack: naive bots that do not follow the "moving target" server replicas automatically, and persistent bots that can follow the moving replicas autonomously with the benign clients. Naïve bots are undermined immediately by the server shuffle, and persistent bots that follow replicas are identified and separated using an intelligent client assignment and tracking scheme during the shuffle.

**Description:**

**Details:** This technique works by shuffling clients from a server under a DDoS attack onto newly instantiated server replicas, and coordinating to maintain service for benign clients while identifying and isolating malicious ones. A client trying to access the protected service goes through the following steps. First, a DNS server directs the client to a cloud domain containing the DDoS defense. Second, the client is directed to a replica server within that cloud by a load balancer, which keeps track of active client/server connections in the cloud domain and tells the assigned replica to whitelist the IP address of the client. Third, the replica server provides the requested service to the client. A centralized coordination server maintains global client/server bindings and coordinates the response to detected DDoS attacks over a dedicated command-and-control channel.

When a DDoS attack is detected, the coordination server triggers the shuffling mechanism, in which clients from attacked servers are shuffled to new server replicas. The scheme for reassigning clients is the output of a greedy algorithm run on the coordination server that maximizes the expected number of benign clients that will be saved in the shuffling round. The algorithm is greedy because it finds the optimal assignment to each new replica server, one target replica at a time, rather than solving the global problem. The shuffling rounds terminate when there is only one available replica left to consider, and it gets all remaining clients assigned to it. Algorithmic details for the greedy algorithm and an optimal one – which is not deployable for real-time use due to its time complexity – are described in [145].

A technique developed after [145] uses a similar reactive migration mechanism for VMs within a cloud and combines it with *proactive* migration, based on the likelihood of a VM becoming attacked [146]. The proactive part of this approach depends on an optimization using parameters that are very difficult to measure or estimate accurately in practice (*e.g.*, the expected attack frequency and idle-period frequency for each VM). Using the proactive scheme with inaccurate inputs could cause a VM to migrate too frequently (creating significant network overhead) or too

infrequently (creating some overhead with little or no security benefit). For these reasons, the proactive mechanism in [146] is not recommended for use outside experimental testbeds.

**Entities Protected:** This technique helps protect cloud servers from DDoS attacks that interrupt services for benign clients.

**Deployment:** This technique is intended for deployment on servers inside a cloud with elastic resources (*i.e.*, to provision and replicate servers on demand).

**Execution Overhead:**

- An optimization problem is solved on the centralized coordination server before shuffling. The time complexity of the greedy algorithm is $O(N \cdot M)$, where $N$ is the total number of clients (benign or bots) and $M$ is the number of persistent bots.
- Performing the redirection during server migration (*i.e.*, moving clients from an attacked server or to new instances) takes time. The authors found that reassigning 60 clients took just under 5 seconds.

**Memory Overhead:**

- The space complexity of the greedy algorithm is $O(P)$, where $P$ is the number of replicas that participate in the shuffling operation.

**Network Overhead:**

- Client redirection overhead during shuffle.

**Hardware Cost:**

- Cost to provision new replica servers in the cloud.

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

Some additional modification may be needed for servers to whitelist client IP addresses as directed by the coordination server.

**Expertise Required to Implement:**

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☒ Custom Programmer (General Knowledge)
- ☒ Custom Programmer (Experiment/Low-Level/Kernel)

Expertise with cloud infrastructure needed for provisioning resources as directed by the shuffling mechanism.

**Expertise Required to Operate:**

- ☒ Seamless
- ☐ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

**Kill-Chain Phases:**

- ☒ Reconnaissance
- ☐ Access
- ☐ Exploit Development
- ☐ Attack Launch
- ☒ Persistence

**Interdependencies:** The technique assumes that DNS servers are well provisioned and are not attacked themselves. The authors assume the attack does not overwhelm the cloud infrastructure itself (*e.g.*, ability to create new replicas). In addition, the defense assumes that a reliable online detector for DDoS attacks is available.

**Weaknesses:** The defense aims to maximize the number of benign clients that receive service during a DDoS attack, but it is not guaranteed to protect service for all benign clients (and does not in the authors' experiments). So despite the moving-target server replicas, the movement can be overcome at least partially by persistent bots. Movement is limited by the number of replica servers available to the defense. The threat model assumes that the cloud infrastructure, DNS

servers, and coordination server are not the targets of an attack, but compromising these could disable movement.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☐ Predict Movement
- ☒ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** The technique makes it more difficult for an attacker to disrupt the protected service to benign clients. In addition to the DDoS resource attack, scanning attacks are mitigated somewhat because servers are replicated and clients may be migrated over time. Therefore, reconnaissance information (particularly, information useful in building a DDoS hit list) may have limited use. That said, the technique does not ensure that persistent bots will not disrupt service to all benign clients. In fact, the authors found that in simulation, the number of shuffling rounds to save 95% of benign clients is 40% higher than the number required to save 80% of benign clients. Therefore, as protection approaches 100% the cost of using the technique increases sharply and may be prohibitive.

**Availability:** The authors implemented the shuffling algorithms and evaluated the technique using simulation. The implementation is not publicly available.

**Additional Considerations:** None

**Proposed Research:** The evaluation in [145] is very theoretical, so a practical evaluation in which a deployed cloud service is targeted by a realistic DDoS attack would help demonstrate its feasibility. Choosing testbed parameters related to resource constraints is important for this defense. For example, if the defense can shuffle every client to its own replica server then the DDoS attack would be fully isolated from benign clients, but that is probably too costly as services scale up to more clients.

**Funding:** DARPA

### 6.15 RECONNAISSANCE DECEPTION SYSTEM

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Scanning

**Details:** This technique [147] is designed to mitigate scanning attacks in a Software-Defined Network (SDN). Malicious reconnaissance is often required for targeted resource attacks or for advanced persistent threats (APTs) that establish persistence and spread laterally in the network. In the threat model, the attacker is an adversary with a foothold on one or more hosts in the network; these compromised hosts are unknown. The defense aims to slow down reconnaissance enough to halt the attack, by identifying and isolating the source of the malicious reconnaissance, before vulnerable hosts are discovered. Not included in the threat model are attacks against the SDN controller or scanning attacks from outside the network, which is addressed by firewalls or intrusion detection systems (IDS).

**Description:**

**Details:** Reconnaissance Deception System (RDS) is a system that simulates a virtual network, which is the only view of the network exposed by insider reconnaissance (including topology, locations of hosts, *etc.*). The goal of the RDS virtual network is to misinform the attacker as much as possible during reconnaissance while causing minimal overhead for benign traffic. RDS performs five maneuvers to achieve this:

1. *Dynamic address translation.* This maneuver uses packet-header rewriting to hide real host addresses. This increases the search space for the scanner because the address space appears much larger.

2. *Route mutation.* This maneuver uses virtual routers that simulate virtual multi-hop paths that hide the real topology.

3. *Vulnerable host placement.* This maneuver uses the previous two maneuvers to simulate virtual topologies and place hosts into virtual subnets.

4. *Honeypot placement.* This maneuver increases the virtual size of the network visible to an attacker and provides decoys that are monitored.

5. *Dynamic detection of malicious flows.* This maneuver uses flow statistics from the SDN switches.

With these techniques deployed, the authors found that RDS delayed malicious network scans up to a factor of 115 by simulating very large virtual networks and strategically placing vulnerable hosts a high "address distance" from the scanning source once it is identified.

Legitimate services in the network could be deceived by the virtual-network views, so nodes that depend on knowing the real network structure (*e.g.*, running scheduling or load-balancing algorithms, or network discovery services) must be identified by an operator and are handled as special cases.

**Entities Protected:** This technique aims to protect hosts in the SDN from insider scanning attackers.

**Deployment:** The technique is deployed in an SDN. Hosts are offered virtual-network views associated with a DHCP lease when they connect to the network.

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- The authors found the system increased response times on average 0.2 ms per packet flow.
- Generating and deploying a new virtual-network view takes on the order of seconds. If done on demand in response to network events, like detected scanning, flows could have additional latency as new views are deployed.

**Hardware Cost:**

- RDS servers and SDN infrastructure, if not already in place.

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

Servers that must know the legitimate network view in order to provide their service must be identified by an operator. The system must be configured so that these nodes are not given the deceptive virtual-network view.

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

**Interdependencies:** RDS's "dynamic detection of malicious flows" feature assumes that benign network nodes are not probing the honeypots or random addresses. The whole technique is dependent on a secured SDN control plane.

**Weaknesses:** Scanning to identify vulnerable hosts is slowed down by this technique, but it is not guaranteed to prevent the attacker from succeeding. Reconnaissance missions that measure link delay to map the network are not mitigated by this technique. Using the SDN control plane to hold new flows and introduce artificial delays is possible, but this would add overhead to legitimate flows and degrade network performance. Attacks against the SDN controller or RDS servers (*i.e.*, deception server, virtual topology generator) could enable the attacker to learn the real network topology, and limit or disable how or when virtual views change. There is some chance that the attacker's foothold host requires the real network view and has been configured to receive it, so in that case RDS provides no protection. This technique does not provide any protection against client-side attacks.

**Types of Weaknesses:**

- ☒ Overcome Movement
- ☒ Predict Movement
- ☒ Limit Movement
- ☒ Disable Movement

**Impact on Attackers:** In general, this technique could slow down an insider reconnaissance attack enough that it halts further attacks. Malicious flows from scanning (*e.g.*, to honeypots that should not have legitimate flows) might be detected and the source mitigated before the attack is finished. However, there is some chance that the reconnaissance succeeds and an attack is launched before the information is made obsolete by generating a new virtual view (because of its integration with the DHCP lease, this might be on the order of hours).

**Availability:** This technique was prototyped by the authors. The open-source prototype is available online at: `https://github.com/deceptionsystem/master`.

**Additional Considerations:** This technique is limited in the protection it provides and adds some latency to flows. The cost of deploying the SDN infrastructure (if it does not already exist) could be high. RDS does not provide any protection if a vulnerable host is actually reached, and it does not protect against client-side attacks. If the SDN is providing other network services on the controller, those could potentially be affected by this technique.

**Proposed Research:** Combining this technique with a hardened SDN control plane would improve its overall security. Another extension is to focus on the gaps in the network defense. The current approach does not offer protection to nodes that run services requiring the real network view; if any of these nodes is compromised, either by being identified quickly during the attack or if one is an initial foothold, then the defense falls apart. There might be ways to have the control/deception plane provide virtual information to these nodes that is crafted to allow nodes to compute correct or correct-enough values without disclosing the real network.

**Funding:** Army Research Laboratory Cyber Security Collaborative Research Alliance

## 6.16 PROVIDING DYNAMIC CONTROL TO PASSIVE NETWORK MONITORING

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource, Scanning, Injection

**Details:** The threat model handled by the technique assumes that malicious hosts are in a network and are trying to initiate network flows to further compromise the network or its endpoints. The technique provides a means for passive monitoring systems, like an intrusion detection system (IDS), to provide a prompt security response when it detects potentially malicious activity. Examples of responses that are possible include **dynamic firewalls** to block traffic; **shunting** flows that are deemed benign to improve network performance and reduce IDS load; **quarantining** local systems that are deemed potentially compromised by blocking all ingress and egress flows; and **quality-of-service** responses that support traffic engineering. Attack techniques can be mitigated as long as they are detectable by the monitoring tools used, *e.g.*, scanning, resource attacks, and sending malicious payloads.

**Description:**

**Details:** This technique presents a *network control framework* [148] that lets passive monitoring systems actuate dynamic network access controls through a unified application program interface (API). The authors build the system atop Bro, an IDS capable of monitoring and responding to the network traffic. When the monitoring system calls a function using the API, response actions (listed above) are triggered by passing response rules to *backends* that are registered with the control framework and are capable of enforcing the actions. For example, an OpenFlow controller in a software-defined network (SDN) might be registered as a backend capable of pushing enforcement rules down to the OpenFlow switches. In this case, quarantining could be enforced by adding flow rules to the OpenFlow switches in front of the affected hosts that drop flows originating from or destined for the hosts. Backends can be any device that can accept response rules generated from API calls and enforce the actions in the network. In addition to OpenFlow, the authors implemented two other backends: 1) an *acld* backend, which is a Unix daemon that functions as a middle-man to firewall against IP addresses, address pairs, ports, etc., and 2) an IDS packet filter internal to Bro itself. When the API is called, backends are passed the resulting response rule until either one can enforce it, or no backend is capable of enforcing it. Protected machines on the network must register with backends and provide the information needed to support enforcement.

**Entities Protected:** This technique protects the network from being attacked from compromised machines.

**Deployment:** This technique is intended for deployment on a network using passive monitoring systems and some backend(s) capable of enforcing dynamic access controls (*e.g.*, OpenFlow

infrastructure). The prototype framework is built on the Bro IDS. It must be deployed with at least one type of backend in order to enforce the response action.

**Execution Overhead:**

- Virtually no overhead beyond using monitoring tools alone. API calls result in rules being generated and handed off to the backends.

**Memory Overhead:**

- OpenFlow rules installed by an OpenFlow backend use memory in the switch tables; however, this overhead is relatively small as long as IDS alerts are infrequent.

**Network Overhead:**

- Potential network service disruptions in response to false-positive API calls.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

Backends and monitoring systems (*e.g.*, Bro) may need to be modified in the source code to incorporate the framework API. Infrastructure components, like OpenFlow switches, must be in place for the OpenFlow backend.

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☐ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☒ Expert Operator

Operations using the framework include all the operations typical of the monitoring system, *e.g.*, verifying IDS alerts, plus responding to false alarms that (in this framework) produce an immediate effect in the network.

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☒ Persistence

The killchain phases depend on what the monitoring system is capable of detecting. Reconnaissance attacks, unauthorized access, and flows aimed at establishing persistence are examples of behavior that could be flagged by a system.

**Interdependencies:** The OpenFlow backend assumes that it can operate without interfering with other SDN applications outside the framework, *e.g.*, installing switch rules that do not conflict.

**Weaknesses:** The coverage of the defense is very dependent on the details of its deployment: the monitoring tools in place and the backends available to enforce the API. Onboarding new network devices could result in policy errors if mistakes are made when registering the device with backends. The monitoring tools must detect events accurately. If malicious events are not detected (false negative), the 'movement' of network access does not actuate. Therefore, attackers can overcome movement by staying under the radar of detection tools. If an attacker knows the framework is in use, it may be possible for them to learn the behaviors that trigger the detectors, and guess which API calls are associated with those events (probably not difficult), enabling attackers to predict the movement.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☐ Limit Movement

☒ Disable Movement

Some movement could be limited by targeted attacks on an individual backend, which enforces some API calls, but this is beyond the scope of the threat model.

**Impact on Attackers:** The technique narrows the gap between when an IDS or other monitoring tool triggers and when a response is applied in the network. Compared to having an administrator manually quarantine a host or update firewalls in response to an alert, an attacker has significantly less time (or none) to finish the attack or evade the response. However, an attacker who is able to stay under the radar of the IDS could evade the system entirely. It is possible that administrators would set less aggressive thresholds for event detection in the framework, since false positives resulting in sudden, incorrect connectivity loss could significantly impede benign network operations.

**Availability:** The authors have a prototype and test scripts available online at: `http://icir.org/johanna/netcontrol`.[1]

**Additional Considerations:** None

**Proposed Research:** One direction for making the framework more easily deployable is automating the process by which devices are onboarded to the backend controls, *e.g.*, using authoritative sensors for network bindings. Another direction is incorporating host context, beyond network traffic, into the decision process that applies the security responses. For example, hosts that are known to have unpatched OS vulnerabilities and exhibit abnormal flows might have a more aggressive traffic threshold for triggering the response compared to a fully-patched host performing the same flows. A similar idea is incorporated into architectures such as Google's BeyondCorp [149] that combine host-context policies with role-based access control.

**Funding:** NSF

[1] Last accessed 5/9/2017.

### 6.17 END POINT ROUTE MUTATION (EPRM)

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource

**Details:** This technique is aimed at mitigating network-availability attacks. In the threat model, the attacker learns a set of critical links, then overwhelms these links through traffic flooding attacks, denying service to the information flows between the endpoints that rely on these network subpaths. The technique addresses this by providing a resilient routing function that performs route mutation efficiently. Other end hosts acting as virtual routers ("peers") are used to increase routing diversity.

**Description:**

**Details:** End Point Route Mutation (EPRM) [150] uses route mutation to defend against distributed denial of service (DDoS) attacks that target links and degrade quality-of-service (QoS) requirements. EPRM formalizes the problem of finding a virtual path for a source-destination pair through a set of peers. It computes a set of paths, then decides on a sequence by which to mutate routes while satisfying two types of constraints: 1) QoS constraints and 2) a resilience constraint. Example QoS constraints mentioned in [150] include:

- Bandwidth: "The available bandwidth of all virtual paths must be greater than the traffic load generated by the source." Otherwise, using a lower bandwidth path still leaves an effect on the service.
- Number of hops: "Number of hops should be limited and the [end to end] delay must be comparable to the delay of the actual physical path of the source-destination pair."

The resilience constraint is one that ensures virtual paths are moving targets for attackers. It says that a sequence of virtual paths should have minimum overlap in the links used in order to increase unpredictability.

The authors note that finding all resilient virtual paths for a single flow is an NP-complete problem, though they are able to use a small upper bound (3) on the number of intermediate peers to make the computation feasible in a real use scenario. They find that even short-length virtual paths using this size bound add enough variance to add protection against persistent attackers.

**Entities Protected:** This technique helps protect network links from DDoS attacks.

**Deployment:** This technique is intended for deployment on end hosts in a network. In general, hosts are assumed to remain in the network (at least for predictable periods of time), since they are configured as virtual routers connected with UDP tunnels. In the implementation described in [150], packet forwarding is set up in kernel space using the Vsys API. The routing paths are computed and installed in the forwarding tables of these hosts.

**Execution Overhead:**

- Computing the set of feasible virtual paths can be expensive and increases as the network size grows, though the authors claim their algorithm is more efficient than solving the general single-source shortest path problem (*e.g.*, using Dijkstra's algorithm).

**Memory Overhead:**

- None

**Network Overhead:**

- Delay overhead associated with virtual paths that have more hops than the actual physical path. In an experiment using ping and a two-peer virtual path, an average of 15 ms was added to the delay.

**Hardware Cost:**

- None

**Modification Costs:**

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

Reconnaissance activities related to delay timing could be thwarted by increased hops in virtual paths.

**Interdependencies:** The technique assumes that peer end hosts exist in the network that are configured as virtual routers.

**Weaknesses:** Details of how routing-table entries are installed into the hosts are vague in [150]. A targeted attack at a component that provides controller functions or synchronization could cause movement of the routing tables and virtual-path sequences to be slowed down or halted, enabling reconnaissance and DDoS attacks against the installed virtual routes. The defense provides no client-side protection, and compromising the host peers and their virtual forwarding tables could derail this scheme. It is also possible that with enough initial reconnaissance, an attacker could run the mutation set and sequence generation algorithms themselves using the real network information, and potentially predict the next virtual paths between a source-destination pair. This could let the attacker target new routes that have the effect of the original attack.

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☒ Limit Movement

☒ Disable Movement

**Impact on Attackers:** The technique makes it more difficult for attackers to execute DDoS attacks against targeted network links. It could make the attack more costly if the attacker tries to

saturate other routes in order to reach the target. That said, an attacker who is able to compromise a host (a peer in EPRM) or leverage known information about the network could potentially overcome the defense.

**Availability:** The authors implemented EPRM in PlanetLab and evaluated it using a virtual network. Currently, the prototype is not publicly available.

**Additional Considerations:** EPRM moves some network functionality into the set of end hosts, which could affect the performance of hosts doing other computations. It is not clear how to best use the technique in networks that having changing topologies (*e.g.*, laptops and mobile devices that come and go, or VMs that are provisioned on demand), where peers could disappear and interrupt routing of other traffic. Peers must also be trusted, which could be an unreasonable assumption depending on the network.

**Proposed Research:** Studying how to incorporate constraints beyond QoS and resilience could be impactful (*e.g.*, routing through peers that may have some trust score). Route mutation might also be used to identify bots that continue predictable attack patterns even when bandwidth is later increased (as Kang et al. [151] demonstrate), which could be incorporated into the technique. Further, reconnaissance attacks that perform mapping based on delays could potentially be mitigated by the technique. These approaches could be evaluated alongside the main goal of relieving DDoS attacks against critical links.

**Funding:** U.S. Army Research Office (ARO) and the Asymmetric Resilient Cybersecurity (ARC) initiative at PNNL

### 6.18 PSI: PRECISE SECURITY INSTRUMENTATION FOR ENTERPRISE NETWORKS

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Resource, Scanning, Injection

**Details:** The threat model handled by the technique is broad and assumes that the goal of attackers is to compromise machines, exfiltrate data, or disrupt service in a network while evading detection and defense mechanisms. It assumes attackers will not infect the defense system (*e.g.*, IDS or SIEM monitoring) or the control plane of the software-defined network (SDN), which is used in the technique. Attackers can use strategies like finding blind spots in the defense systems; adjusting their posture after launching the attack; inducing collateral damage (like degrading network performance by triggering deep packet inspection) through deliberate actions; and overloading defense systems (*e.g.*, denial of service using heavy traffic, though directly compromising systems' logic is out of scope). The technique addresses the threat by providing fine-grained and dynamic security postures, leveraging SDN and network functions virtualization (NFV). Attack techniques can be mitigated as long as they are detectable by the security appliances in the cluster, *e.g.*, scanning, resource attacks, and sending malicious payloads.

**Description:**

**Details:** Precise Security Instrumentation (PSI) [152] is a technique that supports dynamic defenses in an SDN with precision in three dimensions: 1) isolation, so that security policies do not interfere with one another, 2) context, so that policies can be customized to individual devices and their security-related attributes (which the authors call "context"), and 3) agility, so that policies are enforceable at fine-grained time scales, adapting to attacks as they launch and evolve.

PSI works by tunneling a device's traffic to a server cluster ("PSI cluster"), which provides an appropriate security appliance on demand for any type of traffic produced by the device. NFV is used to build small, virtualized appliances (*e.g.*, Bro, Snort) and an SDN within the PSI cluster is used to steer traffic within the cluster to compose these services. The new appliances live on shared commodity hardware to support scalability.

Each device's traffic tunnel is created by having its first-hop edge switch tunnel packets to the gateway switch of the PSI cluster. The controller sees new traffic (Packet-In messages) and installs rules that steer the traffic through the appropriate appliances in the cluster. Events detected by the appliances are passed to the Policy Engine within the cluster, which results in context tags being added to packet headers and determines the context-based forwarding. The isolation and context-based security goals are addressed by having appliances created on demand for individual devices, which have individual policies, when their contexts change. Agility is achieved because new security functions are virtualized and can be deployed rapidly to enforce device-specific policies.

**Entities Protected:** This technique protects the network from being attacked from compromised machines.

**Deployment:** This technique is intended for deployment in an enterprise network. An SDN is used within the PSI cluster to steer traffic.

**Execution Overhead:**

- Some execution overhead on the commodity hardware in the PSI cluster running the control plane and appliance VMs.
- Individual load on controllers is mitigated by automatically scaling out when the arrival rate of Packet-In messages increases.

**Memory Overhead:**

- Some memory overhead on the commodity hardware in the PSI cluster running the appliance VMs and virtual SDN switches.

**Network Overhead:**

- Overhead from appliance monitoring during Packet-In processing.
- Minimal control-plane processing overhead within the cluster due to prefetching and proactive installation of the traffic path through assigned appliances.

**Hardware Cost:**

- Hardware costs related to creating VMs on demand for new virtualized appliances.

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

Source code for virtualized appliances must be modified to expose PSI-added tags in the packet headers. Modifying switch tunneling is needed to send traffic from edge switches to the PSI cluster.

**Expertise Required to Implement:**

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☒ Custom Programmer (General Knowledge)
- ☒ Custom Programmer (Experiment/Low-Level/Kernel)

Tag-based forwarding rules are set up on the SDN switches using custom low-level code. The OpenDaylight SDN controller is also modified to handle events from appliances like Snort.

**Expertise Required to Operate:**

- ☐ Seamless
- ☐ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☒ Expert Operator

An expert operator provides policies via a GUI or domain-specific policy language.

**Kill-Chain Phases:**

- ☒ Reconnaissance
- ☒ Access
- ☐ Exploit Development
- ☒ Attack Launch
- ☒ Persistence

**Interdependencies:** Commodity hardware used to support all the functions of the PSI cluster must be available on the network.

**Weaknesses:** Compromises to the PSI cluster, or the inability to modify packet headers with added context tags from appliances, could limit or disable steering traffic through the security appliances and onto the destination. Naïve implementations could open PSI up to control-plane flooding attacks. Sufficient space is needed in the packet header to add PSI-specific tags for matching against traffic context. As in other monitoring systems, attackers could craft traffic to stay under the radar of installed appliances, overcoming some movement.

**Types of Weaknesses:**

☒ Overcome Movement

☐ Predict Movement

☒ Limit Movement

☒ Disable Movement

To predict movement, an attacker could potentially find a way to leak the traffic-path rules proactively installed in the cluster for the device; additionally, she would need to predict (or evade) the packet-header tags added due to appliance events in that traffic path. The former may be beyond the scope of the threat model.

**Impact on Attackers:** As in [148], the technique narrows the gap between when an IDS or other monitoring tool triggers and when a response is applied in the network. Because the gateway tunnels all traffic through the PSI cluster, successful attacks must succeed despite flowing through the security appliances, which could be difficult for an attacker.

**Availability:** The authors implemented a prototype of PSI, but it is not currently publicly available.

**Additional Considerations:** While the authors of PSI focus on its scalability in an elastic compute environment, the architecture does in fact create choke-points that could be attractive targets to attackers.

**Proposed Research:** The authors describe a use case of PSI where potential false-positive alerts (near the detection threshold) could result in a CAPTCHA challenge rather than an automatic "block" response on traffic. The CAPTCHA could be used to determine whether the flows are actually authorized by the legitimate human user on the device in question. This is a compelling use case, and it raises a separate research question for usable security: how can the traffic data and decision process that PSI uses (with composable appliances and policy) be explained to a human in a way that lets a non-expert verify that PSI is working as expected and that its policies match the intent? There may also be cases where illegitimate traffic is not obviously malicious or unauthorized, so a human cannot easily verify the connections.

**Funding:** NSF, Intel Labs University Research Office

### 6.19 DYNAMIC FLOW ISOLATION

**Defense Category:** Dynamic Networks

**Defense Subcategory:** None

**Threat Model:**

**Attack Techniques Mitigated:** Scanning, Resource

**Details:** This technique [153] was designed to limit the connectivity of endpoints on a network only to the connections that are necessary at any point in time, given a network policy and context. While the technique is in use, host address information gained by an attacker may have limited use because flows to or from the host may be dropped by the network switches depending on the context. This could interrupt command and control sessions, and limit both reconnaissance and lateral movement within the network.

**Description:**

**Details:** Dynamic Flow Isolation (DFI) is a technique that dynamically changes network-level access control in response to changing context in the network (*e.g.*, time of day, security alerts from third-party tools). DFI leverages Software-Defined Networking (SDN) to apply network access policies on-demand to systems on an enterprise network. DFI pushes flow rules to SDN switches that allow, rate limit, or block ingress and egress flows from endpoints; the rules are compiled when new flows are initiated and checked against the current policy, which is updated in response to changing context by policy decision points (PDPs) that process sensor information.

The DFI architecture maintains information about current network bindings of endpoints as well as current policy directives. Using the binding database, a new flow sent to the SDN controller is matched against the current policies during the compilation process, and an OpenFlow rule that enforces the matched policy is formed and sent to the SDN switch. This rule installation occurs before the initial packet actually reaches the controller, because DFI receives the packet from a controller proxy and processes it before sending it northbound to the controller.

Policies in DFI can be informed by a variety of sensors internal and external to the network, such as authentication events, physical location sensors, or antivirus systems. Sensors are assumed not to be networked on the data plane that DFI can restrict, so that policy changes do not interfere with sensor availability. User interfaces that let administrators quarantine hosts or make other access decisions can be hooked into DFI as sensors, allowing for human-in-the-loop control in the framework. PDPs are in charge of handling sensor data and populating the policy database with current policies.

**Entities Protected:** This technique helps hide hosts and servers from scanning and from targeted attacks during times when these machines do not need access, as decided by event-driven decision points.

**Deployment:** This technique requires SDN infrastructure to be at least partially deployed (*i.e.*, controller and one or more SDN switches). Source code may need to be modified to connect desired sensors to PDPs.

**Execution Overhead:**

- None

**Memory Overhead:**

- None

**Network Overhead:**

- Latency overhead occurs when a new flow is sent to the controller by the SDN switch, then enters the DFI access-decision process. This overhead occurs at most once per flow, and is independent of the flow size. If a matching rule for a new flow is already installed on the switch (*e.g.*, if the same user has been active and a previous rule has not expired), there is no overhead as the initial packet and flow are handled by the switch at line speed.

**Hardware Cost:**

- Servers for DFI service components and controller proxy.
- If SDN is not already deployed in the network, one or more SDN switches and a SDN controller.

**Modification Costs:**

☐ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

**Expertise Required to Implement:**

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Experiment/Low-Level/Kernel)

**Expertise Required to Operate:**

☒ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

**Kill-Chain Phases:**

☒ Reconnaissance

☒ Access

☐ Exploit Development

☒ Attack Launch

☒ Persistence

**Interdependencies:** This technique is dependent on having SDN infrastructure at least partially deployed. If it is deployed on a network with both traditional and SDN switches, only flows with SDN switches in their path will have the network-access policy enforced. DFI can be combined with other network-based detection and monitoring systems.

**Weaknesses:** DFI assumes that its components and the SDN switches and controllers are not compromised. It is possible that targeted attacks or lateral movement could succeed if the policy allows connectivity to affected endpoints at that point in time. In these cases, attackers could get lucky in choosing when or where to launch attacks. Alternatively, they could use other reconnaissance attacks to learn about how the decision points change policy in the network, then trigger policy changes via sensor attacks to predict movement. The technique does not protect against client-side attacks, though endpoint monitoring could be used as a sensor for a PDP that restricts access to potentially compromised endpoints. Finally, some sensors that feed into DFI could be targeted by attacks that result in limited movement (*e.g.*, denial of service attacks on sensors, so that decision points do not detect events and change policy).

**Types of Weaknesses:**

☒ Overcome Movement

☒ Predict Movement

☒ Limit Movement

☐ Disable Movement

**Impact on Attackers:** This technique could have varying levels of impact on an attacker depending on what the attacker is trying to accomplish. It could disrupt lateral movement, ongoing command and control attacks, and decrease the likelihood of an attack succeeding. For advanced threats determined to overcome access movement by DFI, this could greatly increase the reconnaissance needed for an attack to succeed.

**Availability:** The authors have prototyped the system and evaluated it on a small pilot network. The prototype is not publicly released.

**Additional Considerations:** How well DFI enforces the principle of least privilege for network access depends on the policy decision points that are implemented. DFI's architecture is extensible to new decision points that could make access control more or less restrictive than in the case piloted by the authors–where end hosts have broad connectivity if and only if they have authorized users logged in, and are otherwise limited to reaching the authentication service.

A system administrator needs to configure the SDN to expire switch rules with the organization's desired security-usability tradeoff in mind. If the time to live (TTL) on a rule is very large, there may be a time gap where access policies have changed but are not yet enforced, potentially leading to unauthorized flows until the rule expires. Setting the TTL to be too small could lead to increased latency due to the switch more frequently sending packets to the DFI controller proxy.

**Proposed Research:** Parts of the DFI architecture could be used for data collection, or for actuating other security controls beyond network access control. For example, instead of just restricting network access, decision points observing changing contexts could also trigger endpoint security tools that might incur too much overhead to run constantly, or turn on aggressive network monitoring tools. Another research direction is exploring how events sensed by the core DFI components themselves (*e.g.*, flows being initiated, or flows that are dropped due to DFI-installed rules) could trigger policy changes. For example, hosts could be used as honeypots to detect unauthorized access using the network, resulting in isolation and monitoring actions for that host and others in the network.

**Funding:** Department of Defense

## REFERENCES


[1] `http://cybersecurity.nitrd.gov/page/moving-target`.

[2] A.D. Keromytis, "Randomized instruction sets and runtime environments past research and future directions," *IEEE Security & Privacy* 7(1), 18–25 (2009).

[3] C.A.P. Enumeration, "Classification (CAPEC)," (2013), https://capec.mitre.org.

[4] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, ACM (2007), pp. 552–561.

[5] H. Okhravi, M. Rabe, T. Mayberry, W. Leonard, T. Hobson, D. Bigelow, and W. Streilein, "Survey of cyber moving target techniques," MIT Lincoln Laboratory, Technical rep. (2013).

[6] P.E. Ammann and J.C. Knight, "Data diversity: An approach to software fault tolerance," *IEEE Transactions on Computers* 37(4), 418–425 (1988).

[7] A. Nguyen-Tuong, D. Evans, J.C. Knight, B. Cox, and J.W. Davidson, "Security through redundant data diversity," in *IEEE International Conference on Dependable Systems and Networks*, IEEE (2008), pp. 187–196.

[8] C. Cadar, P. Akritidis, M. Costa, J.P. Martin, and M. Castro, "Data randomization," Technical Report TR-2008-120, Microsoft Research, 2008., Technical rep. (2008).

[9] M. Christodorescu, M. Fredrikson, S. Jha, and J. Giffin, "End-to-end software diversification of internet services," in *Moving Target Defense*, Springer, pp. 117–130 (2011).

[10] F. Majorczyk and J.C. Demay, "Automated instruction-set randomization for web applications in diversified redundant systems," in *International Conference on Availability, Reliability and Security*, IEEE (2009), pp. 978–983.

[11] S. Son, K. McKinley, and V. Shmatikov, "Diglossia: Detecting code injection attacks with precision and efficiency," in *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, ACM (2013), pp. 1181–1191.

[12] S. Vikram, C. Yang, and G. Gu, "Nomad: Towards non-intrusive moving-target defense against web bots," in *Proceedings of the 2013 IEEE Conference on Communications and Network Security*, IEEE (2013), pp. 55–63.

[13] E. Pattuk, M. Kantarcioglu, Z. Lin, and H. Ulusoy, "Preventing cryptographic key leakage in cloud virtual machines," in *USENIX Security*, USENIX Association (2014), pp. 703–718.

[14] C. Smutz and A. Stavrou, "Preventing exploits in microsoft office documents through content randomization," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, Springer-Verlag New York, Inc. (2015), pp. 225–246.

[15] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, IEEE (2013), pp. 559–573.

[16] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*, IEEE (2014), pp. 575–589.

[17] A.J. O'Donnell and H. Sethu, "On achieving software diversity for improved network secu-rity using distributed coloring algorithms," in *Proceedings of the 11th ACM conference on Computer and communications security*, ACM (2004), pp. 121–131.

[18] T. Fraser, M. Petkac, and L. Badger, "Security agility for dynamic execution environments," in *AFRL-IF-RS-TR-2002-229 Final Technical Report*, DARPA (2002), pp. 1–15.

[19] P.V. Prahbu, Y. Song, and S.J. Stolfo, "Smashing the stack with hydra: The many heads of advanced polymorphic shellcode," *Defcon* 17, 1–20 (2009).

[20] Y. Song, M.E. Locasto, A. Stavrou, A.D. Keromytis, and S.J. Stolfo, "On the infeasibility of modeling polymorphic shellcode," in *Proceedings of the 14th ACM conference on Computer and communications security*, ACM (2007), pp. 541–551.

[21] T. Roeder and F.B. Schneider, "Proactive obfuscation," *ACM Transactions on Computer Systems (TOCS)* 28(2), 4 (2010).

[22] D. Chang, S. Hines, P. West, G. Tyson, and D. Whalley, "Program differentiation," *Journal of Circuits, Systems, and Computers* 21(02) (2012).

[23] T. Andel, L. Whitehurst, and J. McDonald, "Software security and randomization through program partitioning and circuit variation," in *Proceedings of the First ACM Workshop on Moving Target Defense*, ACM (2014), pp. 79–86.

[24] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: transparent code randomization for just-in-time compilers," in *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, ACM (2013), pp. 993–1004.

[25] F. Araujo, K. Hamlen, S. Biedermann, and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM (2014), CCS '14, pp. 942–953.

[26] B. Salamat, A. Gal, and M. Franz, "Reverse stack execution in a multi-variant execution environment," in *Workshop on Compiler and Architectural Techniques for Application Reliability and Security* (2008), pp. 1–7.

[27] T. Jackson, B. Salamat, G. Wagner, C. Wimmer, and M. Franz, "On the effectiveness of multi-variant program execution for vulnerability detection and prevention," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, ACM (2010), p. 7.

[28] T. Jackson, C. Wimmer, and M. Franz, "Multi-variant program execution for vulnerability detection and analysis," in *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, ACM (2010), p. 38.

[29] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proceedings of the 4th ACM European conference on Computer systems*, ACM (2009), pp. 33–46.

[30] M. Franz, "E unibus pluram: massive-scale software diversity as a defense mechanism," in *Proceedings of the 2010 workshop on New security paradigms*, ACM (2010), pp. 7–16.

[31] M. Hafiz and R.E. Johnson, "Security-oriented program transformations," in *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, ACM (2009), p. 12.

[32] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering* 38(1), 54–72 (2012).

[33] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Ap-plications Conference*, ACM (2012), pp. 319–328.

[34] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM (2015), CCS '15, pp. 901–913.

[35] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity," in *Network And Distributed System Security Symposium* (2015), vol. 15.

[36] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE (2016), pp. 431–442.

[37] A. Belay, A. Bittau, A.J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features." in *OSDI* (2012), vol. 12, pp. 335–348.

[38] C. Kil, C. Jun, J. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Computer Security Applications Conference* (2006).

[39] M. Frantzen and M. Shuey, "Stackghost: Hardware facilitated stack protection." in *USENIX Security Symposium* (2001), vol. 112.

[40] G.C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27(3), 477–526 (2005).

[41] H. Shacham, M. Page, B. Pfaff, E.J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, ACM (2004), pp. 298–307.

[42] T. Durden, "Bypassing pax aslr protection," *Phrack magazine* 59(9), 9 (2002).

[43] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, ACM (2009), pp. 1–8.

[44] A. Sotirov and M. Dowd, "Bypassing browser memory protections: Setting back browser security by 10 years," *Blackhat USA* (2008).

[45] V. Pappas, M. Polychronakis, and A. Keromytis, "Dynamic reconstruction of relocation information for stripped binaries," in A. Stavrou, H. Bos, and G. Portokalidis (eds.), *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, Springer International Publishing (2014), pp. 68–87.

[46] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened ASLR on Android," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, IEEE (2014), pp. 424–439.

[47] E. Berger and B. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Im-plementation*, ACM (2006), pp. 158–168.

[48] G. Novark and E.D. Berger, "Dieharder: securing the heap," in *Proceedings of the 17th ACM conference on Computer and communications security*, ACM (2010), pp. 573–584.

[49] A. Smirnov and T.c. Chiueh, "A portable implementation framework for intrusion-resilient database management systems," in *International Conference on Dependable Systems and Networks*, IEEE (2004), pp. 443–452.

[50] P. Sousa, A.N. Bessani, M. Correia, N.F. Neves, and P. Verissimo, "Resilient intrusion tolerance through proactive and reactive recovery," in *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, IEEE (2007), pp. 373–380.

[51] S. Sidiroglou, O. Laadan, A.D. Keromytis, and J. Nieh, "Using rescue points to navigate software recovery," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*, IEEE (2007), pp. 273–280.

[52] M.C. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee, "Enhancing server availability and security through failure-oblivious computing." in *OSDI* (2004), vol. 4, pp. 21–21.

[53] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan, "Preventing overflow attacks by memory randomization," in *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, IEEE (2010).

[54] C. Giuffrida, A. Kuijsten, and A. Tanenbaum, "Enhanced opearing system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Conference on Security Symposium*, USENIX Association (2012), pp. 40–40.

[55] G. Zhu and A. Tyagi, "Protection against indirect overflow attacks on pointers," in *Proceedings of the Second IEEE International Information Assurance Workshop*, IEEE (2004), pp. 97–106.

[56] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu, "Code shredding: Byte-granular randomization of program layout for detecting code-reuse attacks," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ACM (2012), pp. 309–318.

[57] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ACM (2012), pp. 157–168.

[58] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the IEEE Symposium on Security and Privacy* (2013), pp. 574–588.

[59] "Polyverse," `https://polyverse.io`.

[60] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson, "ILR: Where'd my gadgets go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, IEEE (2012), pp. 571–585.

[61] S.H. Kim, L. Xu, Z. Liu, Z. Lin, W.W. Ro, and W. Shi, "Enhancing software dependability and security with hardware supported instruction address space randomization," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE (2015), pp. 251–262.

[62] V. Pappas, M. Polychronakis, and A. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, IEEE (2012), pp. 601–615.

[63] "Morphisec," `https://www.morphisec.com/`.

[64] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *Proceedings of the 23rd USENIX Security Symposium*, USENIX Association (2014), pp. 443–447.

[65] L. Davi, C. Liebchen, A.R. Sadeghi, K.Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming." in *22nd Annual Network and Distributed Systems Security Symposium* (2015).

[66] A. Bittau, A. Belay, A. Mashtizadeh, D. Maziéres, and D. Boneh, "Hacking blind," in *Proceedings of the IEEE Symposium on Security and Privacy* (2014), pp. 227–242.

[67] T. Petsios, V.P. Kemerlis, M. Polychronakis, and A.D. Keromytis, "Dynaguard: Armoring canary-based protections against brute-force attacks," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ACM (2015), pp. 351–360.

[68] D. Williams-King, G. Gobieski, K. Williams-King, J. Blake, X. Yuan, P. Colp, M. Zheng, V. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *12th USENIX Symposium on Operating Systems Design and Implementation*, USENIX Association (2016), pp. 367–382.

[69] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to make ASLR win the clone wars: Runtime re-randomization," in *23rd Annual Network and Distributed Systems Security Symposium* (2016).

[70] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *23rd Annual Network and Distributed Systems Security Symposium* (2016).

[71] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, et al., "Address-oblivious code reuse: On the effectiveness of leakage-resilient diversity," (2017).

[72] M. Sun, J. Lui, and Y. Zhou, *Blender: Self-randomizing address space layout for android apps*, Springer International Publishing, pp. 457–480 (2016).

[73] S.J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a TRaP: Table randomization and protection against function-reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM (2015), pp. 243–255.

[74] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *IEEE Symposium on Security and Privacy*, IEEE (2015), pp. 763–780.

[75] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM (2015), pp. 256–267.

[76] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM (2014), pp. 54–65.

[77] K.Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," in *IEEE Symposium on Security and Privacy (SP)*, IEEE (2016), pp. 954–968.

[78] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehen-sive protection from stack-based memory error vulnerabilities for binaries." in *22nd Annual Network and Distributed Systems Security Symposium* (2015).

[79] V. Mohan, P. Larsen, S. Brunthaler, K.W. Hamlen, and M. Franz, "Opaque control-flow integrity." in *22nd Annual Network and Distributed Systems Security Symposium* (2015), vol. 26, pp. 27–30.

[80] K. Lu, C. Song, B. Lee, S.P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM (2015), pp. 280–291.

[81] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.R. Sadeghi, and H. Okhravi, "Address-oblivious code reuse: On the effectiveness of leakage-resilient diversity," in *24th Annual Network and Distributed Systems Security Symposium* (2017).

[82] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM (2015), pp. 268–279.

[83] H. Hu, S. Shinde, S. Adrian, Z.L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE Symposium on Security and Privacy (SP)*, IEEE (2016), pp. 969–986.

[84] H. Okhravi, "Tracer: Timely randomization applied to commodity executables at runtime," Cyber Security Division Transition to Practice Technology Guide (2016), `https://www.dhs.gov/sites/default/files/publications/CSD%20TTP%20FY16%20Tech%20Guide.pdf`.

[85] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: Defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Com-puter Security Applications Conference* (2010), pp. 49–58.

[86] S. Checkoway, L. Davi, A. Dmitrienko, A.R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Com-puter and communications security*, ACM (2010), pp. 559–572.

[87] Y. Weiss and E.G. Barrantes, "Known/chosen key attacks against software instruction set randomization," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd An-nual*, IEEE (2006), pp. 349–360.

[88] A.N. Sovarel, D. Evans, and N. Paul, "Where's the feeb? the effectiveness of instruction set randomization." in *Usenix Security* (2005).

[89] G.F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib (c)," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, IEEE (2009), pp. 60–69.

[90] T. Wei, T. Wang, L. Duan, and J. Luo, "Insert: Protect dynamic code generation against spraying," in *2011 International Conference on Information Science and Technology*, IEEE (2011), pp. 323–328.

[91] W. Hu, J. Hiser, D. Williams, J. Filipi, A. Davidson, D. Evans, J. Knight, A. Nguyen-Tuong, and J. Rowanhill, "Secure and practical defense against code-injection attacks using software," in *Proceedings of the 2nd International Conference on Virtual Execution Environments*, ACM (2006), VEE '06, pp. 2–12.

[92] K. Scott and J. Davidson, "Safe virtual execution using software dynamic translation," in *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, IEEE (2002), pp. 209–218.

[93] A. Nguyen-Tuong, A. Wang, J.D. Hiser, J.C. Knight, and J.W. Davidson, "On the effectiveness of the metamorphic shield," in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ACM (2010), pp. 170–174.

[94] G.S. Kc, A.D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and communications security*, ACM (2003), pp. 272–280.

[95] X. Jiang, H. Wangz, D. Xu, and Y.M. Wang, "RandSys: Thwarting code injection attacks with system service interface randomization," in *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems* (2007).

[96] L.Q. Nguyen, T. Demir, J. Rowe, F. Hsu, and K. Levitt, "A framework for diversifying windows native apis to tolerate code injection attacks," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, ACM (2007), pp. 392–394.

[97] G. Portokalidis and A.D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *Proceedings of the 26th Annual Computer Security Applications Con-ference*, ACM (2010), pp. 41–48.

[98] E. Barrantes, D. Ackley, S. Forrest, and D. Stefanovi, "Randomized instruction set emulation," *ACM Transactions on Information Systems Security* 8(1), 3–40 (2005).

[99] E.G. Barrantes, D.H. Ackley, T.S. Palmer, D. Stefanovic, and D.D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*, ACM (2003), pp. 281–289.

[100] S.W. Boyd, G.S. Kc, M.E. Locasto, A.D. Keromytis, and V. Prevelakis, "On the general applicability of instruction-set randomization," *IEEE Transactions on Dependable and Secure Computing* 7(3), 255–270 (2010).

[101] S. Boyd and A. Keromytis, *SQLrand: Preventing SQL Injection Attacks*, Springer, pp. 292–302 (2004).

[102] Z. Liang, B. Liang, L. Li, W. Chen, Q. Kang, and Y. Gu, "Against code injection with system call randomization," in *International Conference on Networks Security, Wireless Communications and Trusted Computing*, IEEE (2009), vol. 1, pp. 584–587.

[103] M. Petkac and L. Badger, "Security agility in response to intrusion detection," in *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, IEEE (2000), pp. 11–20.

[104] D. Williams, W. Hu, J.W. Davidson, J.D. Hiser, J.C. Knight, and A. Nguyen-Tuong, "Security through diversity: Leveraging virtual machine technology," *IEEE Security & Privacy* 7(1) (2009).

[105] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz, "Runtime defense against code injection attacks using replicated execution," *IEEE Transactions on Dependable and Secure Computing* 8(4), 588–601 (2011).

[106] D.A. Holland, A.T. Lim, and M.I. Seltzer, "An architecture a day keeps the hacker away," *ACM SIGARCH Computer Architecture News* 33(1), 34–41 (2005).

[107] C. Taylor and J. Alves-Foss, "Diversity as a computer defense mechanism," in *Proceedings of the 2005 workshop on New security paradigms*, ACM (2005), pp. 11–14.

[108] R.A. Maxion, "Use of diversity as a defense mechanism," in *Proceedings of the 2005 workshop on New security paradigms*, ACM (2005), pp. 21–22.

[109] K. Beznosov and P. Kruchten, "Towards agile security assurance," in *Proceedings of the 2004 workshop on New security paradigms*, ACM (2004), pp. 47–54.

[110] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity." in *Usenix Security* (2006), vol. 6, pp. 105–120.

[111] H. Okhravi, A. Comella, E. Robinson, and J. Haines, "Creating a cyber moving target for critical infrastructure applications using platform diversity," *International Journal of Critical Infrastructure Protection* 5(1), 30–39 (2012).

[112] B.J. Min and J.S. Choi, "An approach to intrusion tolerance for mission-critical services using adaptability and diverse replication," *Future Generation Computer Systems* 20(2), 303–313 (2004).

[113] S.B.E. Raj and G. Varghese, "Analysis of intrusion-tolerant architectures for web servers," in *2011 International Conference on Emerging Trends in Electrical and Computer Technology*, IEEE (2011), pp. 998–1003.

[114] A. Saidane, V. Nicomette, and Y. Deswarte, "The design of a generic intrusion-tolerant architecture for web servers," *IEEE Transactions on dependable and secure computing* 6(1), 45–58 (2009).

[115] A.K. Bangalore and A.K. Sood, "Securing web servers using self cleansing intrusion tolerance (scit)," in *Dependability, 2009. DEPEND'09. Second International Conference on*, IEEE (2009), pp. 60–65.

[116] Y. Huang, D. Arsenault, and A. Sood, "Incorruptible system self-cleansing for intrusion tolerance," in *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, IEEE (2006), pp. 493–496.

[117] D. Arsenault, A. Sood, and Y. Huang, "Secure, resilient computing clusters: self-cleansing intrusion tolerance with hardware enforced security (scit/hes)," in *The Second International Conference on Availability, Reliability and Security*, IEEE (2007), pp. 343–350.

[118] M. Crouse and E.W. Fulp, "A moving target environment for computer configurations using genetic algorithms," in *CProceedings of the 4th Symposium on onfiguration Analytics and Automation*, IEEE (2011), pp. 1–7.

[119] D.J. John, R.W. Smith, W.H. Turkett, D.A. Cañas, and E.W. Fulp, "Evolutionary based moving target cyber defense," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ACM (2014), pp. 1261–1268.

[120] Y. Huang and A.K. Ghosh, "Introducing diversity and uncertainty to create moving attack surfaces for web services," in *Moving Target Defense*, Springer, pp. 131–151 (2011).

[121] Y. Huang and A.K. Ghosh, "Automating intrusion response via virtualization for realizing uninterruptible web services," in *Network Computing and Applications, 2009. NCA 2009. Eighth IEEE International Symposium on*, IEEE (2009), pp. 114–117.

[122] "Software protection initiative," `http://www.spi.dod.mil/lipose.htm`.

[123] Z. Wang and R.B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM SIGARCH Computer Architecture News*, ACM (2007), vol. 35, pp. 494–505.

[124] S.J. Moon, V. Sekar, and M.K. Reiter, "Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM (2015), pp. 1595–1606.

[125] M. Thompson, N. Evans, and V. Kisekka, "Multiple os rotational environment: An implemented moving target defense," in *Proceedings of the 7th International Symposium on Resilient Control Systems*, IEEE (2014).

[126] M. Thompson, M. Muggler, M. Marilyne, and I. Moses, "Dynamic application rotation environment for moving target defense," in *Resilience Week*, IEEE (2016), pp. 17–26.

[127] V. Varadarajan, T. Ristenpart, and M.M. Swift, "Scheduler-based defenses against cross-vm side-channels." in *Usenix Security* (2014), pp. 687–702.

[128] Y. Zhang and M. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, ACM (2013), pp. 827–838.

[129] D. Kewley, R. Fink, J. Lowry, and M. Dean, "Dynamic approaches to thwart adversary intelligence gathering," in *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, IEEE (2001), vol. 1, pp. 176–185.

[130] J. Michalski, C. Price, E. Stanton, E. Lee, K. Chua, Y. Wong, and C. Tan, "Network security mechanisms utilizing dynamic network address translation," (2002).

[131] J. Li, P.L. Reiher, and G.J. Popek, "Resilient self-organizing overlay networks for security update delivery," *IEEE Journal on Selected Areas in Communications* 22(1), 189–202 (2004).

[132] H. Moniz, N.F. Neves, M. Correia, and P. Verissimo, "Randomized intrusion-tolerant asynchronous services," in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, IEEE (2006), pp. 568–577.

[133] S. Antonatos, P. Akritidis, E.P. Markatos, and K.G. Anagnostakis, "Defending against hitlist worms using network address space randomization," *Computer Networks* 51(12), 3471–3490 (2007).

[134] E. Al-Shaer, "Toward network configuration randomization for moving target defense," in *Moving Target Defense*, Springer, pp. 153–159 (2011).

[135] J.D. Touch, G.G. Finn, Y.S. Wang, and L. Eggert, "DynaBone: Dynamic defense using multi-layer internet overlays," in *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, IEEE (2003), vol. 2, pp. 271–276.

[136] "AFRL resources," Personal communication.

[137] "CryptoniteNXT," `http://www.cryptonitenxt.com/`. Last accessed: 2017-04-06.

[138] E. Al-Shaer, Q. Duan, and J. Jafarian, *Random Host Mutation for Moving Target Defense*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 310–327 (2013).

[139] J. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: Transparent mov-ing target defense using software defined networking," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ACM (2012), pp. 127–132.

[140] Y.B. Luo, B.S. Wang, X.F. Wang, X.F. Hu, and G.L. Cai, "TPAH: A universal and multi-platform deployable port and address hopping mechanism," in *2015 International Conference on Information and Communications Technologies*, IET (2015), pp. 1–6.

[141] J. Jafarian, E. Al-Shaer, and Q. Duan, "Spatio-temporal address mutation for proactive cyber agility against sophisticated attackers," in *Proceedings of the First ACM Workshop on Moving Target Defense*, ACM (2014), pp. 69–78.

[142] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, ACM (2013), pp. 413–424.

[143] M. Albanese, A. De Benedictis, S. Jajodia, and K. Sun, "A moving target defense mechanism for manets based on identity virtualization," in *Proceedings of the 2013 IEEE Conference on Communications and Network Security*, IEEE (2013), pp. 278–286.

[144] Y. Li, R. Dai, and J. Zhang, "Morphing communications of cyber-physical systems towards moving-target defense," in *Proceedings of the 2014 IEEE International Conference on Com-munications*, IEEE (2014), pp. 592–598.

[145] Q. Jia, H. Wang, D. Fleck, F. Li, A. Stavrou, and P. W., "Catch me if you can: A cloud-enabled ddos defense," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE (2014), pp. 264–275.

[146] S. Debroy, P. Calyam, M. Nguyen, A. Stage, and V. Georgiev, "Frequency-minimal mov-ing target defense using software-defined networking," in *2016 International Conference on Computing, Networking and Communications*, IEEE (2016), pp. 1–6.

[147] S. Achleitner, T. La Porta, P. McDaniel, S. Sugrim, S. Krishnamurthy, and R. Chadha, "Cyber deception: Virtual networks to defend insider reconnaissance," in *Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats*, ACM (2016), pp. 57–68.

[148] J. Amann and R. Sommer, "Providing dynamic control to passive network security monitoring," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, Springer-Verlag New York, Inc. (2015), pp. 133–152.

[149] R. Ward and B. Beyer, "Beyondcorp: A new approach to enterprise security," *login* 39, 5–11 (2014).

[150] U. Rauf, F. Gillani, E. Al-Shaer, M. Halappanavar, S. Chatterjee, and C. Oehmen, "Formal approach for resilient reachability based on end-system route agility," in *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, ACM (2016), pp. 117–127.

[151] M. Kang, V. Gligor, and V. Sekar, "SPIFFY: Inducing cost-detectability tradeoffs for persistent link-flooding attacks," in *23rd Annual Network and Distributed Systems Security Symposium* (2016).

[152] T. Yu, S.K. Fayaz, M. Collins, V. Sekar, and S. Seshan, "Psi: Precise security instrumentation for enterprise networks," in *24th Annual Network and Distributed Systems Security Symposium* (2017).

[153] R. Skowyra and D. Bigelow, "Dynamic flow isolation: Adaptive access control to protect networks," Cyber Security Division Transition to Practice Technology Guide (2016), `https://www.dhs.gov/sites/default/files/publications/CSD%20TTP%20FY16%20Tech%20Guide.pdf`.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| 17 January 2018 | Technical Report | |

**4. TITLE AND SUBTITLE**

Survey of Cyber Moving Targets
Second Edition

**5a. CONTRACT NUMBER**
FA8721-05-C-0002 and/or FA8702-15-D-0001

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**

B.C. Ward, S.R. Gomez, R.W. Skowyra, D. Bigelow, J.N. Martin, J.W. Landry, H. Okhravi

**5d. PROJECT NUMBER**
2805

**5e. TASK NUMBER**
271

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02421-6426

**8. PERFORMING ORGANIZATION REPORT NUMBER**

TR-1228

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Department of Defense

**10. SPONSOR/MONITOR'S ACRONYM(S)**
DoD

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION / AVAILABILITY STATEMENT**

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

**13. SUPPLEMENTARY NOTES**

**13. ABSTRACT**

This survey provides an overview of different cyber moving-target techniques, their threat models, and their technical details. A cyber moving-target technique refers to any technique that attempts to defend a system and increase the complexity of cyber attacks by making the system less homogeneous, static, or deterministic. This survey describes the technical details of each technique, identifies the proper threat model associated with the technique, as well as its implementation and op-erational costs. Moreover, this survey describes the weaknesses of each technique based on the current proposed attacks and bypassing exploits, and provides possible directions for future research in that area.

**15. SUBJECT TERMS**

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | 19b. TELEPHONE NUMBER *(include area code)* |
| Unclassified | Unclassified | Unclassified | Unclassified | 333 | |