Chapter 8

CREATING A CYBER MOVING TARGET FOR CRITICAL INFRASTRUCTURE APPLICATIONS*

Hamed Okhravi, Adam Comella, Eric I. Robinson, Stephen Yannalfo, Peter W. Michaleas, and Joshua Haines

Abstract Despite the significant amount of effort that often goes into securing critical infrastructure assets, many systems remain vulnerable to advanced, targeted cyber attacks. This paper describes the design and implementation of the Trusted Dynamic Logical Heterogeneity System (TALENT), a framework for live-migrating critical infrastructure applications across heterogeneous platforms. TALENT permits a running critical application to change its hardware platform and operating system, thus providing cyber survivability through platform diversity. TALENT uses containers (operating-system-level virtualization) and a portable checkpoint compiler to create a virtual execution environment and to migrate a running application across different platforms while preserving the state of the application (execution state, open files and network connections). TALENT is designed to support general applications written in the C programming language. By changing the platform on-the-fly, TALENT creates a cyber moving target and significantly raises the bar for a successful attack against a critical application. Experiments demonstrate that a complete migration can be completed within about one second.

Keywords: Cyber moving target, Cyber survivability, Platform heterogeneity, Virtualization

∗ This work is sponsored by the Department of Defense under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

1. Introduction

Critical infrastructure systems are an integral part of the national cyber infrastructure. The power grid, oil and gas pipelines, utilities, communications systems, transportation systems, and banking and financial systems are examples of critical infrastructure systems. Despite the significant amount of effort and resources used to secure these systems, many remain vulnerable to advanced, targeted cyber attacks. The complexity of these systems and their use of commercial-of-the-shelf components often exacerbate the problem.

Although protecting critical infrastructure systems is a priority, recent cyber incidents [9, 16] have shown that it is imprudent to rely completely on the hardening of individual components. As a result, attention is now focusing on game-changing technologies that achieve mission continuity during cyber attacks. In fact, the U.S. Air Force Chief Scientists report on technology horizons [5] mentions the need for "a fundamental shift in emphasis from 'cyber protection' to 'maintaining mission effectiveness' in the presence of cyber threats" as a way to build cyber systems that are inherently intrusion resilient. Moreover, the White House National Security Councils progress report [2] mentions a "moving target" – a system that moves in multiple dimensions to foil the attacker and increase resilience – as one of the Administrations three key themes for its cyber security research and development strategy.

This paper describes the design and implementation of the Trusted Dynamic Logical Heterogeneity System (TALENT), a framework for live-migrating critical applications across heterogeneous platforms. In mission-critical systems, the mission itself is the top priority, not individual instances of the component. TALENT can help thwart cyber attacks by live-migrating the mission-critical application from one platform to another. Also, by dynamically changing the platform at randomly-chosen time intervals, TALENT creates a cyber moving target that places the attacker at a disadvantage and increases resilience. This means that the information collected by the attacker about the platform during the reconnaissance phase becomes ineffective at the time of attack.

TALENT has several design goals:

- Heterogeneity at the instruction set architecture level, meaning that applications should run on processors with different instruction sets.
- Heterogeneity at the operating system level.
- Preservation of the state of the application, including the execution state, open files and sockets. This is an important property in mission-critical systems because simply restarting the application from scratch on a different platform may have undesirable consequences.
- Working with a general-purpose system language such as C. Much of TALENT's functionality is straightforward to implement using a platform-

independent language like Java because the Java Virtual Machine provides a sandbox for applications. However, many commodity and commercial-of-the-shelf software systems are developed in C. Restricting TALENT to a Java-like language would limit its use.

TALENT must provide operating system and hardware heterogeneity while preserving the state and environment despite the incompatibility of binaries between different architectures. TALENT addresses these challenges using: (i) operating-system-level virtualization (container-based operating system) to sandbox the application and migrate the environment including the filesystem, open files and network connections; and (ii) portable checkpoint compilation to compile the application for different architectures and migrate the execution state across different platforms.

TALENT is novel in several respects. TALENT is a heterogeneous platform system that dynamically changes the instruction set and operating system. It supports the seamless migration of critical applications across platforms while preserving their states. Neither application developers nor operators require prior knowledge about TALENT; TALENT is also application agnostic. Other closely-related solutions either lose the internal state of applications or are limited to specific types of applications (e.g., web servers). The TALENT implementation is optimized to reduce the migration time – the current prototype completes the migration of state and environment in about one second. To the best of our knowledge, TALENT is the first realization of a cyber moving target through platform heterogeneity.

2. Threat Model

The TALENT threat model assumes there is an external adversary who is attempting to exploit a vulnerability in the operating system or in the application binary in order to disrupt the normal operation of a mission-critical application. For simplicity and on-the-fly platform generation, a hypervisor (hardware-level virtualization) is used. The threat model assumes that the hypervisor and the system hardware are trusted. We assume that the authenticity of the hypervisor is checked using hardware-based cryptographic verification (e.g., TPM) and that the hypervisor implementation is free of bugs. We also assume that the operating-system-level virtualization logic is trusted. However, the rest of the system (including the operating system and applications) is not trusted and may contain vulnerabilities and malicious logic.

We also assume that, although an attack may be feasible against a number of different platforms (operating system/architecture combinations), there exists a platform that is not susceptible to the attack. This means that not all the platforms are vulnerable. Our primary goal is to migrate a mission-critical application to a different platform at random time intervals when a new vulnerability is discovered or when an attack is detected. Attacks can be detected using various techniques that are described in the literature (e.g., [10]).

Heterogeneity at different levels can mitigate attacks. Application-level heterogeneity protects against binary- and architecture-specific exploits, and untrusted compilers. Operating-system-level heterogeneity mitigates kernel-specific attacks, operating-system-specific malware and persistent operating system attacks (rootkits). Finally, hardware heterogeneity can thwart supply chain attacks, malicious and faulty hardware, and architecture-specific attacks. It is important to note that TALENT is by no means a complete defense against all these attacks. Instead, it is designed to enhance survivability in the presence of platform-specific attacks using dynamic heterogeneity.

3. Design

TALENT uses two key concepts, operating-system-level virtualization and portable checkpoint compilation, to address the challenges involved in using heterogeneous platforms, including binary incompatibility and the loss of state and environment.

3.1 Virtualization and Environment Migration

Preserving the environment of a critical infrastructure application is an important goal. The environment includes the filesystem, configuration files, open files, network connections and open devices. Note that many of the environment parameters can be preserved using virtual machine migration. However, virtual machine migration can only be accomplished using a homogeneous operating system and hardware. Indeed, virtual machine migration is not applicable because it is necessary to change the operating system and hardware while migrating a live application.

TALENT uses operating-system-level virtualization to sandbox an application and migrate the environment.

3.1.1 OS-level virtualization. In operating-system-level virtualization, the kernel allows for multiple isolated user-level instances. Each instance is called a container (jail or virtual environment). The method was originally designed to support fair resource sharing, load balancing and cluster computing applications. This type of virtualization can be thought of as an extended `chroot` in which all resources (devices, filesystem, memory, sockets, etc.) are virtualized.

Note that the major difference between operating-system-level virtualization and hardware-level virtualization (e.g., Xen and KVM) is the semantic level at which the entities are virtualized (Figure 1). Hardware-level hypervisors virtualize disk blocks, memory pages, hardware devices and CPU cycles,

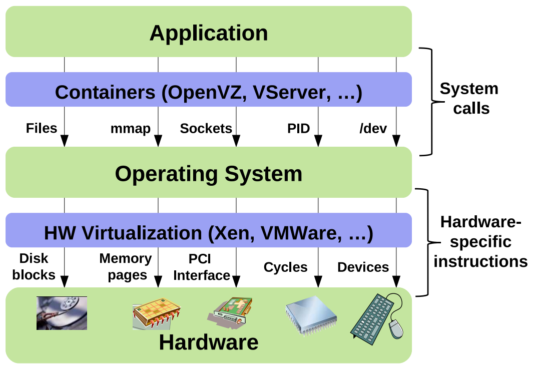

Figure 1. OS-level and hardware-level virtualization approaches

whereas operating-system-level virtualization works at the level of filesystems, memory regions, sockets and kernel objects (e.g., IPC memory segments and network buffers). Hence, the semantic information that is often lost in hardware virtualization is readily available in operating-system-level virtualization. This makes operating-system-level virtualization a good choice for applications where semantic information is needed, for example, when monitoring or sandboxing at the application level.

3.1.2 Environment Migration. As discussed above, TALENT uses operating-system-level virtualization to migrate the environment of a critical application. When migration is requested (as a result of a malicious activity or a periodic migration), TALENT migrates the container of the application from the source machine to the destination machine. This is done by synchronizing the filesystem of the destination container with the source container. Since the operating system keeps track of open files, the same files are opened in the destination. Because this information is not available at the hardware virtualization level (due to the semantic gap between files and disk blocks), additional techniques must be implemented to recover the information (e.g., virtual machine introspection). On the other hand, this information is readily available in TALENT.

Network connections can be virtualized in three ways: second layer, third layer and socket virtualization. These terms come from the OpenVZ documentation [18]. Virtualizing a network at the second layer means that each container has its own IP address, routing table and loopback interface. Third layer virtualization implies that each container can access any IP address/port and that sockets are isolated using the namespace. Socket virtualization means that each container can access any IP address/port and that sockets are isolated using filtration.

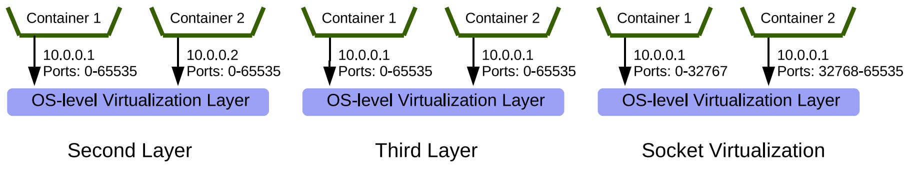

Figure 2. Different network virtualization approaches

Figure 2 shows the different network virtualization approaches for two containers. In socket virtualization, the port numbers are divided between the containers, whereas in third layer virtualization, the entire port range is available to every container. TALENT uses second layer virtualization in order to be able to migrate the IP address of a container.

To preserve network connections during migration, the IP address of the containers virtual network interface is migrated to the new container. Then, the state of each TCP socket (`sk_buff` of the kernel) is transferred to the destination. The network migration is seamless to the application, and the application can continue sending and receiving packets on its sockets. In fact, our evaluation example shows that the state of an SSH connection is preserved during the migration.

Many operating-system-level virtualization frameworks also support IPC and signal migration. In each case, the states of IPC and signals are extracted from the kernel data structures and migrated to the destination. These features are supported in TALENT through the underlying virtualization layer.

3.2 Checkpointing and Process Migration

Migrating the environment is only one step in backing up the system because the state of running programs must also be migrated. To do this, a method to checkpoint running applications must be implemented. After all the checkpointed program states are saved in checkpoint files, the state is migrated by simply mirroring the filesystem.

3.2.1 Requirement. Checkpointing in TALENT must meet certain requirements.

- **Portability:** Checkpointed programs should be able to move back and forth between different architectures and operating systems in a heterogeneous computing environment.

- **Transparency:** Heavy code modification should not be required to existing programs in order to introduce proper checkpointing.

- **Scalability:** Checkpointed programs may be complex and may handle large amounts of data. Checkpointing should be able to handle such programs without affecting system performance.

A portable checkpoint compiler (PCC) can help meet the portability requirement. Figure 3 illustrates the portable checkpoint compilation process, which allows compilation to occur independently on various operating system/architecture pairs. The resulting executable program, including the inserted checkpointing code, functions properly on each platform on which it was compiled.

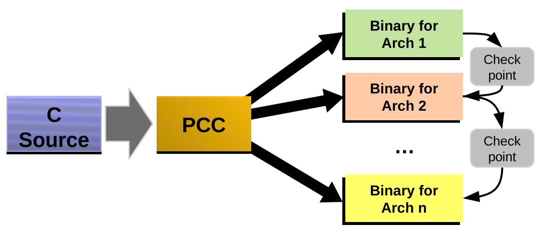

Figure 3. Portable checkpoint compilation

Transparency is obtained by performing automatic code analysis and checkpoint insertion. This prevents the end user from having to modify their code to indicate where checkpointing should be performed and what data should be checkpointed.

Scalability is obtained in two ways. First, the frequency of checkpointing bottlenecks in the checkpointing process is controlled. Second, through the use of compressed checkpoint file formats, the checkpoints themselves remain as small as possible even as the amount of data processed by the program increases.

3.2.2 Variable Level Checkpointing. There are two possible approaches for checkpointing a live process: data segment level checkpointing (DSLC) and variable level checkpointing (VLC). Note that DSLC and VLC are different types of portable checkpoint compilers.

In DSLC [25], the entire state of the process including the stack and heap are dumped into a checkpoint file. DSLC preserves the entire state of a process, but since the checkpoint file contains platform specific data such as the stack and heap, this approach suffers from a lack of portability.

VLC [8], on the other hand, stores the values of restart-relevant variables in the checkpoint file. Since the checkpoint file only contains portable data, VLC is a good candidate for migration across heterogeneous platforms. In order to construct the entire state of the process, VLC must re-execute the non-portable portions of the code. The non-portable portions refer to the platform-dependent values stored in the stack or heap, not the variables.

To perform VLC, the code is precompiled to find restart-relevant variables. These variables and their memory locations are then registered in the checkpointing tool. When checkpointing, the process is paused and the values of the memory locations are dumped into a file. The checkpointing operation must occur at safe points in the code to generate a consistent view.

At restart, the memory of the destination process is populated with the desired variable values from the checkpoint file. Some portions of the code are re-executed in order to construct the entire state.

A simple example involving a factorial computation is presented to illustrate VLC operation. Of course, TALENT is capable of handling much more complicated code bases. The factorial code is shown in Figure 4. For simplicity, the code incorporates a main function with no inputs.

```
int main(int argc, char **argv)
{
   int fact;
   double curr;
   int i;

   fact = 20;
   curr = 1;
   for(i=1; i<=fact; i++)
   {
      curr = curr * i;
   }
   printf("%d factorial is %f",
          fact, curr);
   return 0;
}
```

Figure 4. A simple program to compute the factorial of 20

Figure 5 illustrates the VLC markup of the factorial program. All calls to the checkpointing tool are shown with pseudo-function calls with `VLC_` prefixes. First, the checkpointer is initialized. Then, the variables to be tracked and checkpointed are registered with the tool. In the example, the variables `fact`, `curr` and `i` have to be registered. The actual checkpointing must occur inside the loop after each iteration. When the loop is done, there is no need to track the variables any longer, so they are unregistered. Finally, the environment is torn down before the `return`. Note that for transparency and scalability, the code markup has been done automatically and prior to compilation.

3.2.3 Checkpoint Portability.

The checkpoint file itself must have a portable format to achieve portability across heterogeneous platforms.

```
int main(int argc, char **argv)
{
   int fact;
   double curr;
   int i;
   VLC_INITIALIZE();
   fact = 20;
   curr = 1;
   VLC_REGISTER_VARIABLES(fact, curr, i);
   for(i=1; i<=fact; i++)
   {
      VLC_PERFORM_CHECKPOINT();
      curr = curr * i;
   }
   VLC_UNREGISTER_VARIABLES(fact, curr, i);
   printf("%d factorial is %f",
          fact, curr);
   VLC_TEAR_DOWN();
   return 0;
}
```

Figure 5. Variable level checkpointing of the factorial program

Storing the checkpoint in a simple binary file can result in incompatibility if the destination platform has different "bitness" (32 vs. 64 bits) or endianness (little vs. big). Thus, the checkpoint file format has to be portable.

TALENT uses the HDF5 format [3] through the precompiler checkpointing tool. HDF5 is an open, versatile data model that can represent complex data objects. It is also portable in that it can represent various types of bitness and endianness. Like XML, HDF5 is self-describing. Unlike XML, HDF5 uses a binary format that allows for the efficient parsing of data.

Figure 6 illustrates the complete migration process. First, the environment of the application is migrated using container migration. Then, the application itself is checkpointed and resumed on the destination platform. Heterogeneous platforms are illustrated using different colors in the figure. The application box on the destination platform shows a different binary of the same application that is compiled for the platform.

4. Implementation

TALENT is implemented using the OpenVZ container-based operating system and the CPPC portable checkpoint compiler.

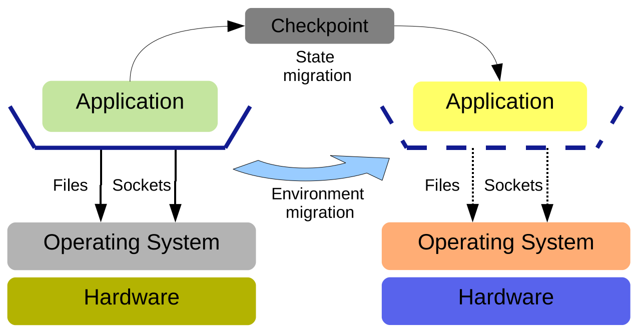

Figure 6. TALENT migration process

4.1 Environment Migration

Several operating-system-level virtualization implementations are available, including OpenVZ [18] and LXC [4] for Linux, Virtuozzo [1] for Windows, and Jail [22] for FreeBSD. We have chosen UNIX-like operating systems as our platform. We have also chosen OpenVZ as the container because of its ease of use, stable code and support for second layer network virtualization. In particular, we have used OpenVZ version 2.6.27 and have patched it into different kernels; KVM [13] has been used as the underlying hypervisor.

We have implemented and tested TALENT on Intel Xeon 32-bit, Intel Core 2 Quad 64-bit and AMD Opteron 64-bit processors. Also, we have tested TALENT on the Gentoo, Fedora (9, 10, 11 and 12), CentOS (4 and 5), Debian (4 and 5), Ubuntu (8 and 9) and SUSE (10 and 11) operating systems. In total, we have tested 37 combinations.

For environment migration, we do not use the OpenVZ live migration feature because it migrates the processes within the container and causes binary incompatibility. Instead, we migrate the environment by freezing the container, synchronizing the filesystem, migrating the virtual network interface and transferring buffers, IPC and signals. We then substitute the binary of the application built for the destination processor. Note that TALENT can also be implemented across more diverse operating systems such as Windows using the Virtuozzo [1] container. In this case, migrating complex environment features such as signals and IPC requires more effort because they have to be mapped correctly to the destination platform.

4.2 Process Migration

Given the desired requirements enumerated in Section 8.3.2.1, TALENT employs the Controller/Precompiler for Portable Checkpointing (CPPC) [21] to save the state of a running program. CPPC is a VLC precompiler implementation. It is capable of storing the program state of a running program in a format that is operating system and hardware independent (HDF5), and then correctly restarting the program on a different platform using the previously-stored state.

CPPC is a compiler-assisted checkpointing program that involves four execution phases:

- **Compiling the Code:** The code is compiled on each platform independently.
- **Configuring the Run:** The preferences for checkpointing during a run are configured.
- **Checkpointing:** The run is started and checkpointing of the state occurs automatically.
- **Restarting the Run:** The checkpoint (after being migrated) is resumed on a new platform.

4.2.1 Compiling the Code. CPPC can compile traditional C and Fortran 77 code. It compiles unmodified source code and the programmer does not need to have any knowledge of checkpointing. CPPC automatically determines how and where to checkpoint a program. Section 8.4.3 shows an example of how this is done.

CPPC interfaces with the user code as a precompiler. It uses the Cetus compiler infrastructure [19] to determine the semantic behavior of a program in order to decide where to place checkpointing directives. Once this is determined, the code is re-factored with checkpointing function calls. The re-factored code can then be compiled using a traditional compiler such as `cc` or `gcc`.

4.2.2 Configuring the Run. CPPC requires a configuration file in order to run with checkpointing. This file specifies the parameters used for checkpointing, including the frequency of checkpointing, the number of checkpoints to be stored and their storage locations. Although a default file is provided, a user may wish to configure the file based on the expected behavior of the program. For example, the frequency of checkpointing can be increased for critical applications that change frequently so as to capture the most recent state; or the frequency can be decreased for slowly-changing programs to avoid bottlenecks when writing files.

Run parameters can be changed directly in the configuration file by modifying the appropriate values or by using command line options when starting a run. Typically, a program will have a suitable configuration specified in the configuration file. However, a user may override the configuration to obtain different behavior by entering a new value for a parameter via the command line. The configuration file can be stored in text or XML formats.

4.2.3 Checkpoint.

Checkpoints are stored in a file using the HDF5 format [3]. Since this format is deployed on many platforms, checkpoint files can be stored in a manner that is compatible across a range of architectures and operating systems. Additionally, a CRC-32-based algorithm is supported to verify the integrity of checkpoint files.

As stated above, checkpointing is done automatically. The user may change the rate at which checkpointing is performed via the configuration file or the command line. Compiler options also allow programmers to manually specify where checkpointing should occur by adding `#pragma` directives to the source code. Directives also exist for other CPPC functionality such as indicating code that should be run upon restart for re-initializing data not stored in memory, or for other initialization tasks such as restarting the message passing interface.

4.2.4 Restarting from a Checkpoint.

After a run has been started and a checkpoint has been recorded, it is possible to restart the run from the last recorded checkpoint. This is done on the same platform or on a different platform. “Jump” statements are added in the original code to the locations of the checkpoints. Based on the checkpoint file, the jump locations are known upon restart. These jump states are ignored during the initial run so that the program is executed as if no changes were introduced. In addition to jumping to the appropriate starting location, the checkpoint file contains information about variable values within the program. These are loaded upon restart to ensure that the program resumes in the same state it was upon checkpointing.

4.3 Code Example

We revisit the factorial code in Figure 4 to illustrate the operation of the checkpointer.

First, the code is automatically converted to a markup code using `#pragma` directives to specify where special CPPC content should be inserted. Figure 7 shows the markup for the factorial code. Note that the variables to be checkpointed are registered with CPPC.

Next, CPPC uses the markup in Figure 7 to create a final version of the code that the C compiler can understand. The final code is not shown here due to its length. However, the concepts involved in generating the code are straightforward. For each checkpoint, line labels are inserted to mark the locations of the checkpoints. The labels are tracked using an array that is populated when CPPC is initialized. Each label is assigned a unique ID based on its location in

```
int main(int argc, char **argv)
{
   int fact;
   double curr;
   int i;
   #pragma cppc init
   fact = 20;
   curr = 1;
   #pragma cppc register
      ( fact, curr, i )
   for(i=1; i<=fact; i++)
   {
      #pragma cppc checkpoint
      curr = curr * i;
   }
   #pragma cppc unregister
      ( fact, curr, i )
   printf("%d factorial is %f",
          fact, curr);
   #pragma cppc shutdown
   return 0;
}
```

Figure 7. The cppc markup of the factorial program

the array. When a call to checkpoint is made, the appropriate ID is also stored in the checkpoint file.

When the program is restarted, the call to initialize re-populates the registered values that were in memory from the previous run. The code then jumps to the appropriate checkpoint label. This is achieved by using a "goto" command to jump to the line in the line label array referenced by the ID stored in the checkpoint file. From here, the program proceeds as normal, continuing to checkpoint at the indicated locations in the program.

5. Evaluation

We have developed a test application to evaluate the performance of TALENT. The application contains 2,000 lines of C code and a GUI developed using wxWidgets [23]. The graphical output of the application is sent to a remote machine via an SSH connection. Upon receiving a migration request, the application and its GUI are migrated from a Gentoo/Intel Xeon 32-bit machine to an Ubuntu 10.04.1/AMD Opteron 64-bit machine using environment migration and checkpointing.

The original migrations took a long time (about a minute), so we decided to time the individual elements of migration. After breaking down the delays,

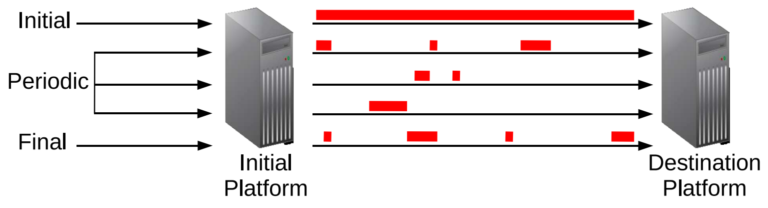

Figure 8. Optimized filesystem synchronization model

we discovered that synchronizing the filesystem took 98.7% of the migration time. This is not surprising because, during a migration, the entire filesystem available to the container must be copied to the destination. As a result, we decided to focus on optimizing the filesystem synchronization.

In the optimized version, the filesystem is synchronized with the destination once before the migration occurs. The synchronization is subsequently performed at periodic intervals by sending the differences to the destination.

Figure 8 presents the optimized synchronization model. We chose 30 seconds as the synchronization interval. When a migration is requested, only the differences are sent to the destination. This simple optimization reduces the environment migration time to about one second.

Figure 9 shows the performance of TALENT with and without optimization. Note that quota and configuration refer to checking the resource quotas (CPU, disk, memory, etc.) assigned to each container and verifying the platform configurations, respectively. If the optimization is enabled, then network traffic to the destination platform has to be strictly limited to filesystem updates to prevent the attacker from performing a reconnaissance of the destination.

During the migration, the graphical output at the remote machine disappears for about two seconds. When the migration is completed, the graphical output reappears on the remote terminal (now running on the second platform) without any user intervention because the state of the SSH connection is preserved.

6. Related Work

Several data segment level [25, 11] and variable level process migration techniques [8] have been proposed in the literature. These methods are often used in high performance and cluster computing systems for load balancing and fault tolerance.

Virtual machine migration [12] has also been proposed as a cluster administration technique for load balancing, online maintenance and power management. However, it requires a homogeneous architecture and operating system in order to preserve state.

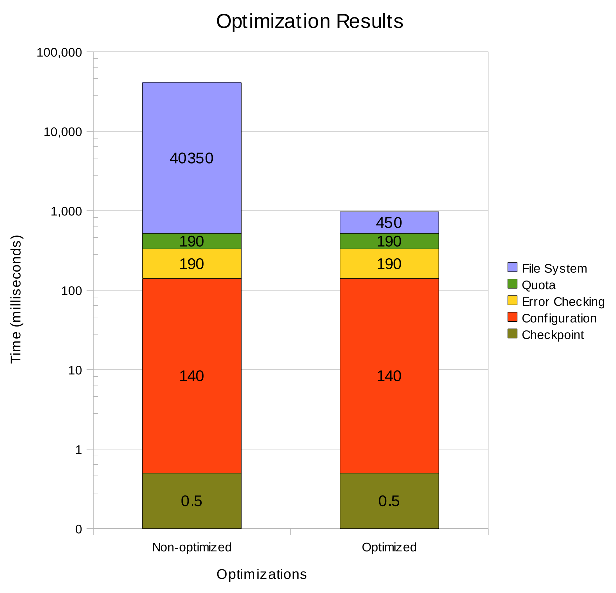

Figure 9. TALENT's performance with and without optimization

The Self-Cleansing Intrusion Tolerance (SCIT) Project [24] is closely related to TALENT. It migrates an application across different virtual machines to reduce the exposure time. Our work differs from SCIT in a number of ways. First, TALENT preserves the state of the application. SCIT-web server [6] and SCIT-DNS [26] preserve the session information and DNS master file and keys, respectively, but not the internal state of the application. Second, TALENT uses heterogeneous platforms for migration. The designers of SCIT mention the use of diverse operating systems and memory images to further confuse an attacker, but the current implementation uses the same operating system with the same configuration [6]. Finally, TALENT is designed to support general critical infrastructure applications and is not limited to a specific server.

The Resilient Web Service (RWS) Project [14] uses a virtualization-based web server system that detects intrusions and periodically restores them to a pristine state. Its successor, RWS-Moving Target (RWS-MT) [15] plans to use diversity to create a moving target, but it only focuses on web servers as the critical application. In addition, both systems lose the state of the web server. Certain forms of server rotation have been proposed by Blackmon and Nguyen [7] and by Rabbat, et al. [20] in an attempt to achieve high availability servers.

7. Conclusions

The TALENT system provides dynamic, heterogeneous platforms for critical infrastructure applications. It creates a cyber moving target that offers resilience in the face of platform-specific cyber attacks. To the best of our knowledge, TALENT is the first heterogeneous platform solution that preserves the internal state of a general application.

The current TALENT prototype is focused on providing high availability. There is no guarantee that the migrated state (persistent or ephemeral) is not already corrupted. In future work, we plan to extend TALENT by adding sanitization and recovery capabilities. This would provide integrity guarantees for an application under attack. We also plan to augment TALENT with an attack detection engine that can trigger migration. Finally, we plan to integrate TALENT with an assessment framework based on attack graphs [17] so that the destination platform can be selected based on formal vulnerability and reachability analysis.

Note that the opinions, interpretations, conclusions and recommendations in this paper are those of the authors and are not necessarily endorsed by the U.S. Government.

8. Acknowledgements

This work was sponsored by the U.S. Department of Defense under Air Force Contract FA8721-05-C-0002.

References

[1] Clustering in parallels virtuozzo-based systems. White paper, Parallels, 2009.

[2] Cybersecurity Progress after President Obama's Address. The White House National Security Council, July 2010.

[3] HDF4 Reference Manual. The HDF Group, February 2010. ftp://ftp.hdfgroup.org/HDF/Documentation/HDF4.2.5/HDF425_RefMan.pdf.

[4] LXC Man Pages, 2010. http://lxc.sourceforge.net/index.php/about/man/.

[5] Report on Technology Horizons: A Vision for Air Force Science & Technology During 2010–2030. AFST-TR-10-01-PR, United States Air Force Chief Scientist, May 2010.

[6] A. K. Bangalore and A. K. Sood. Securing web servers using self cleansing intrusion tolerance (scit). In *Proceedings of the 2009 Second International Conference on Dependability*, pages 60–65, 2009.

[7] S. Blackmon and J. Nguyen. High-availability file server with heartbeat. *System Admin, The Journal for UNIX and Linux Systems Administration*, 10(9), 2001.

[8] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs. *SIGPLAN Not.*, 38:84–94, June 2003.

[9] R. H. Brown. Stuxnet worm causes industry concern for security firms, October 2010. http://www.masshightech.com/stories/2010/10/18/daily19-Stuxnet-worm-causes-industry-concern-for-security-firms.html.

[10] G. Carl, G. Kesidis, R. R. Brooks, and S. Rai. Denial-of-service attack-detection techniques. *IEEE Internet Computing*, 10(1):82–89, 2006.

[11] Y. Chen, K. Li, and J. Plank. Clip: A checkpointing tool for message passing parallel programs. In *Supercomputing, ACM/IEEE 1997 Conference*, pages 33 – 33, 1997.

[12] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05*, pages 273–286. USENIX Association, 2005.

[13] I. Habib. Virtualization with kvm. *Linux J.*, 2008(166):8, 2008.

[14] Y. Huang and A. Ghosh. Automating intrusion response via virtualization for realizing uninterruptible web services. In *Network Computing and Applications, 2009. NCA 2009. Eighth IEEE International Symposium on*, pages 114 –117, 2009.

[15] Y. Huang, A. Ghosh, and T. Bracewell. A security evaluation of a novel resilient web serving architecture: Lessons learned through industry/academia collaboration. In *International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 188–193, 2010.

[16] ICS-CERT. Control system internet accessibility. Ics-alert-10-301-01, Industrial Control Systems Cyber Emergency Response Team, October 2010.

[17] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer. Modeling modern network attacks and countermeasures using attack graphs. In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference*, pages 117–126, 2009.

[18] K. Kolyshkin. Virtualization in linux. White paper, OpenVZ, September 2006.

[19] S. Lee, T. A. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *16th Intl. Workshop on Languages and Compilers for Parallel Computing*, pages 539–553, 2003.

[20] R. Rabbat, T. McNeal, and T. Burke. A high-availability clustering architecture with data integrity guarantees. In *IEEE International Conference on Cluster Computing*, pages 178–182, 2001.

[21] G. Rodríguez, M. J. Martín, P. González, J. Touri no, and R. Doallo. Cppc: a compiler-assisted tool for portable checkpointing of message-passing applications. *Concurrency and Computation: Practice and Experience*, 22(6):749–766, 2010.

[22] E. Sarmiento. Securing freebsd using jail. *Sys Admin*, 10(5):31–37, 2001.

[23] J. Smart, K. Hock, and S. Csomor. *Cross-Platform GUI Programming with wxWidgets (Bruce Perens Open Source)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[24] A. Sood. Intrusion tolerance to mitigate attacks that persist. In *Proceedings of the Secure and Resilient Cyber Architectures Conference*, SRCA'10, McLean, Virginia, 2010. MITRE.

[25] G. Stellner. Cocheck: checkpointing and process migration for mpi. In *Proceedings of The 10th International Parallel Processing Symposium (IPPS '96)*, pages 526 –531, Apr. 1996.

[26] A. S. Yih Huang, David Arsenault. Incorruptible self-cleansing intrusion tolerance and its application to dns security. *AJournal of Networks*, 1(5):21–30, September/October 2006.