State-Capture, Check-pointing and Host Migration of Linux Processes

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin-La Crosse

La Crosse, Wisconsin

by

Tylor E. Arndt

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

January, 2017

State-Capture, Check-pointing and Host Migration of Linux Processes

By Tylor E. Arndt

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

_____ _____

Dr. Thomas Gendreau, Ph.D — Date

Examination Committee Chairperson

_____ _____

Dr. Kenny Hunt, Ph.D — Date

Examination Committee Member

_____ _____

Dr. Samantha S. Foley, Ph.D — Date

Examination Committee Member

Abstract

Arndt, Tylor, E., "State-Capture, Check-pointing and Host Migration of Linux Processes", Master of Software Engineering, January 2017, Dr. Thomas Gendreau, Ph.D.

This manuscript describes a tool that implements a methodology to capture the state of a Linux process. Once captured, a snapshot represents the state of an application. This state that can be stored as a checkpoint for later restoration or examination on the same local or a remote host. For example, should environmental inference, such as hardware or power failure terminate the application prematurely it could be restored or a snapshot which has been taken of application that subsequently crashed could be restored for debugging. This tool can be used pro-actively to transfer a process to another host machine, allowing for schedulable maintenance or easy load balancing.

Acknowledgements

First and foremost of my acknowledgments, are the several faculty members of the University Wisconsin La Crosse Computer Science department who have been instrumental in facilitating this work. I would like to thank my undergrad advisor, Mark Headington for kindling a deep love for systems software in me. This is a passion I am happy to express both in this work and in my professional occupation.

I am very grateful for the patience and wisdom of my capstone advisor Thomas Gendreau. After beginning my MSE Capstone I had the good fortune to be hired at a startup doing my dream-job, then have said startup be acquired by a Fortune 500 company, and go on to help build a large thriving systems software team within the acquire. This fortunate series of events was not conducive to expedient completion of this project, and Dr. Gendreau's kindness, advice and endurance have been above and beyond what I could have expected.

I would like to also give special thanks to Kenny Hunt, who provided me much needed encouragement when life's distractions where leading me to consider abandoning this effort.

Last, but not least, I need to thank my beautiful wife, Arika Arndt, whose quiet encouragement, support and patience have been invaluable to this endeavor.

Table of Contents

1. Introduction

1.1 Background, Terminology and Project Motivations

When this project began, the author was disappointed by the rise in popularity of host level virtualization by industry [1] in lieu of operating system level abstractions . Host level virtualization technology has existed in the mainframe world since early in the computer revolution, the pioneering example being IBM's CP-40 time sharing system in 1967[2], and became a standard feature of the z/OS operating system. It was not until the late 1990's that companies such as VMware brought this technology to commodity PC server hardware.

The term *virtual machine (VM)* is used for two related technologies that implement the same abstract goals of execution emulation. One class of virtual machine is the *process virtual machine*, in which an application's runtime provides portability and flexibility facilities (such as a portable executable instruction representation, bytecode). One example of this kind of VM would be the Java Virtual Machine (JVM)[3]. While *process virtual machines* share the *virtual machine* nomenclature, they are not the form of virtual machine discussed here.

In the context of this project, virtual machine to refers to *system virtual machine*, specifically an isolation container that visualizes a physical (hardware) ISA (industry standard architecture). This use of the term is most similar to the concept of a VM as "an efficient, isolated duplicate of a real machine" as introduced by Gerald J. Popek and Robert P. Goldberg in their 1974 article "Formal Requirements for Virtualizable Third Generation Architectures"[4].

Host-level virtualization was developed early in computing when operating systems were quite immature. These early systems often lacked memory management units (MMUs), and

if they existed they were quite primitive and usually were overlay or segmentation-based. The lack of modern style paged memory made host-level virtualization more feasible for these early systems than implementing process isolation. While MMU-based process isolation came to be the prevailing process isolation mechanism, the failure of system designers and integrators in the late 1990's to produce adequately secure operating systems provided ample incentive for resurrection of this technology for the PC server platform by companies such as VMware[5]. While building host-level virtualization for the commodity Linux /Windows server ecosystem, these vendors brought a number of features such a *checkpointing, host-migration, live-migration* and *multi-host lock-step execution* that previously had only been available to mainframes.

Check-pointing: Is the ability to take a snapshot at a point in time of an executing program so that it can be restored back to this previous state in the future. This is useful for long running and expensive computations or decreasing application initialization time at startup (by taking a post-startup snapshot).

Host-migration: Related to Check-pointing. The ability to stop a VM (or application) and transfer its state to a new host and then resume execution. The amount of time the computation is halted varies considerably with implementation and environmental constraints, such as the local network bandwidth and latency.

Live-migration: Is similar to host-migration, but it is completed within a predetermined minimal amount of time where execution is not active. Ideally, the execution pause would be small enough to be similar to the suspension delay suffered by a local processes while waiting for the operating system scheduler to resume execution in a preemptive multitasking system. This minimal pause time is accomplished by running the state capture and migration concurrently with program execution, identifying already transferred state that mutates, and continuously and progressively transmitting state deltas until the remaining unsynchronized state delta is small enough for a final transmission within the targeted execution suspension

duration.

This project is based on the point of view that the challenge of accommodating many applications on a shared modern system can be addressed without having to resort to the older course grained approach virtualizing the entire system. Furthermore, having to do so represented a failure of operating systems to fulfill their original design objectives and promises of user-space isolation and hardware abstraction. Host-level virtualization is wasteful because without complex compression and de-duplication strategies it leads to duplication in memory and disk of operation system assets and other shared system resources. Additionally, host-level virtualization adds another layer of abstraction in the form of the hypervisor which also has significant overhead [6]. Ideally operating systems would improve upon the security/isolation failures of the 1990's and would be enhanced to also provide the state management facilities previously exclusive to mainframes such as host-migration and live-migration. This view has recently been validated by the drastic rise in popularity of new stronger process isolation provided by containerized computing as a replacement for virtual machines [7].

1.2 Project Objectives and Description

The Linux and BSD kernel communities have begun increase process isolation and decrease the risks security defects by adding support for *containers* [8] which are operating system provided opt-in higher isolation execution contexts for processes. While these containerization efforts seek to address the failure of operating systems to fulfill their isolation promises, it does not address the lack of check-pointing and process migration facilities found both in older mainframe technology and newer host-level virtualization solutions.

The goal of this project is to bring some of these process state manipulation related facilities to Linux which is the most popular server operating system in the world [9][10]. Specifically

providing Linux tooling to manage process check-pointing and migration where the primary project objectives, with live-migration being a future goal that should not be precluded by design decisions.

2.0 Research, Theory and Discovery

2.1 Identification and Definition of Program State

Any discussion of capturing and manipulating program state must be prefaced with identification and definition of what is encompassed by program state. The following table is a summary of the process state that needs to be considered:

State Component	**Component's Location**
Process tree owner	Program Configuration file/argument
PID/GID	Kernel-space
Virtual memory permissions	Kernel-space (Page table, also in CPU)
Open file handles	Kernel-space
Semaphore states	Kernel-space
Outstanding signals	Kernel-space
Performance counters	Kernel-space
Open socket/pipes	Kernel-space
Multiple Threads	Kernel-space
Flushing open files to disk	User and Kernel space
Program heap	User-space
Program stack	User-space
Program register states	CPU
IP Address	System configuration
MAC Address	System configuration

Table 1: Components of process state and corresponding location

It is not necessary to capture all of the above state, rather the extent to which state is captured determines the system facilities an application may utilize while reamining eligible for state capture by the implemented solution. In an effort to limit scope and establish criteria of the MVP (minimal viable product) the following subset of state was selected for

capture in the initial implementation of the application:

State Component	**Component's Location**
Process tree owner	Program Configuration file/argument
PID/GID	Kernel-space
Virtual memory permissions	Kernel-space (Page table, also in CPU)
Open file handles	Kernel-space
Flushing open files to disk	User and Kernel space
Program heap	User-space
Program stack	User-space
Program register states	CPU

Table 2: Components of process state selected for capture in the MVP

This subset of state will allow most simple single-threaded applications performing local I/O to be captured and establishes a foundation to add increasingly complex layers of functionality in later iterations.

2.2 State Capture and Convergence Algorithms

Fundamentally there are two methods of capturing process state that differ primarily in the liveness of the application during the state capture process. The most straight-forward approach is to halt the execution of the target process while the state of the process is captured; alternatively the process can be allowed to make progress while state is being captured. Forward progress is beneficial for the check-pointing use-case; it allows applications to be check-pointed in a manner that is transparent to users because it does not effect application availability. Forward progress is also a natural prerequisite for live-migration for similar reasons of user transparency and application availability.

In the simpler halted process method, check-pointing and migration operations begin similarly:

1. Supervisor attaches to target process and halts it
2. Process state is captured
3. State is serialized to persistent storage or remote sink

At this point there is a divergence in methodology for the check-pointing use-case as opposed to migration. A check-pointing operation will resume the target process and allow it to continue, whereas, a migration operation will transfer the captured state to a remote host and terminate the local process. The remote cooperating host will then use the state that was captured in the checkpoint snapshot that as been transferred to resume the process locally. If the process does not make forward progress during the capture operation, the first state delta is also the final delta thus state convergence for any migration operation is guaranteed. This approach is sometimes referred to as the stop-and-copy approach [11].

In the more complicated case of check-pointing or migrating a live process, it is useful to refer to the very similar work that has been done with live-migration in virtual machines. Two methods are commonly employed: one is a push-based state transfer and the other a pull-based transfer [11]. In the pushed-based scheme, state is continuously transferred until the remaining state is small enough to be transferred within a fixed threshold. This involves potentially retransmitting state, such as memory pages, as the application state change. In the case of host migrations, the amount of data actually transfered can be limited by transmitting delta of atomic units (again such as pages or disk blocks) rather than the entire unit and can be further minimized by applying compression. For check-pointing, to reduce the size of the resulting snapshot there may need to be a deduplication operation where the original state item has any state deltas applied before the final persistent result is recorded.

Raw theoretical 4KB page transfer rates over network as compared to memory interfaces		
Interface	Megabytes/Second	Pages/Second
10 Mbit Ethernet	1.25	320
100 Mbit Ethernet	12.5	3200

1 Gbit Ethernet	125	3200
10 Gbit Ethernet	1250	32000
40 Gbit Ethernet	5000	128000
Haswell 2-ch DDR3 Write [12]	22000	5632000
Haswell 3-ch DDR4 Write [12]	48000	12288000

Table 3: Common I/O interfaces and their data transfer rates

Ultimately these optimizations are insufficient to guarantee local and remote state convergence in cases where the application's state mutates faster than the state can be transfered. Since L2 cache throughput often exceeds 12 GB/s and memory bandwidth often exceeds 6 GB/s, and both of which are much higher than even compressed network throughput, there is a high likelihood of this eventuality in practice for certain classes of users. To mitigate this, the process supervising the state transfer may throttle the amount of state mutation the target processed is allowed, which will have the direct impact of reducing the amount of forward progress from that which would take place under normal operation. It is also essential in such a scheme that the local capturing and remote receiving/supervising processes establish their effective state transfer rates, either through user configuration or ideally, experimental determination, at the onset or during state transfer. Once the remaining state is small enough to be transferred within a user-established threshold, the checkpoint can be finalized or the local process halted, and the remote clone can be started.

The alternative to halted-process and push-based state transfer is a pull-based method. In the pull approach a minimal amount of state is transferred. Interestingly this state, such as the page table mappng, CPU registers and open file handles, is often the state which is transfered last in the push-based method. After this initial minimal state is transfered, the remote clone begins execution, and the original target process is halted. The remote clone supervisor then handles page faults in the cloned application and retrieves the required state on demand from the target process supervisor on the original host. This method has the benefit of enacting instantaneous transfer but, even with optimization such as prefetching, suffers from latency proportional to the latency and bandwidth of the link between original and cloned process supervisors. This approach is only applicable to migrations as opposed to

check-pointing operations.

While the halted-process method is mutually exclusive of the push and pull methods, the push and pull methods can potentially be combined into a hybrid approach. The following what a network-aware push-based algorithm might look like:

State migration takes place in five phases:

1. *Initialization* - This is when the source and destination daemons that are involved in the migration establish communication and verify parameters, network latency and bandwidth.
2. *Alpha pass (pass 1)* – The first iteration of the state synchronization algorithm.
3. *Pass N* – One or more intermediary iterations of the state synchronization algorithm.
4. *Omega pass (last pass)* - The final iteration of the state synchronization algorithm.
5. *Finalization* – This is when execution of the target process tree is halted on the source system and execution begins on the target system. Also during this time, any external notifications are issued. For example proxy ARP commands need to be distributed to routers and switches to guarantee TCP/IP traffic is delivered to the destination host and not to the source host.

2.3 Discussion of Check-pointing and Migration versus Fault Tolerance

Live-migration is a proactive action taken to allow planned maintenance of a host providing a service or load balancing . Live-migration allows another host to assume the execution of the service without any interruption in the availability of the service. In contrast to live-migration, fail-over is a remedial action taken to mitigate a hardware failure or other unexpected outage, by allowing a service to resume execution on another secondary host after the primary host has failed.

An important question during the research and discovery phase of this project in regards to framing scope was to understand the extent to which state capture for the sake of multi-host state migration was similar to multi-host fault tolerant execution. However, based on industry software implementations of fault tolerence by vendors such as VMware and Microsoft, it became apparent during the literature review that different state replication techniques are typically used for fail-over vs. live-migration [13].

The state migration technique outlined in the project proposal consisted of a constant replication of an application's delta state between a primary and secondary host. This approach is inconsistent with the method VMware, and other industry leaders, use to implement their fail-over technology and their live-migration technology.

Leading software vendors typically attain fault-tolerance by executing the services in CPU lock step on the primary and secondary host. In this way no state needs to be transferred between hosts; yet, all CPU utilization is duplicated. Furthermore, all writes to disk made by the secondary host are disregarded.

This distinction means that while state migration and fault-tolerance technologies are related, fault-tolerance is technologically more divergent than the migration stop-and-copy and live-copy methods are to each-other.

2.4 Local Migration Concerns

System-call interdiction is the act of intercepting a system call to observe or modify system-call arguments before they are passed to the operating system, and then once again during the kernel-space user-space context switch when the result of the system call is returned to the calling application. One very simple use-case for this is the getpid system-call. When a process is restored from a check-point snapshot or migrated (and cloned on a new host), it is

likely that the old process identifier (PID) is no longer available on the system. By employing system-call interdiction to inject logic, we can guarantee that the application in the restored execution context continues to see its original PID when calling getpid by substituting the result returned by the system call with the old PID value.

When an application is migrated or resumed, its previous files and sockets can be re-opened; however, the re-created resource's file descriptor (aka handle, a number) will no longer be consistent between the application's expectation and the OS. To rectify this inconsistency, the code supervising the resume/migration must interdict system calls made by the application and substitute old descriptor values the new values for the corresponding resource. This substitution is possible by having the supervising restore process maintain a mapping of old to new descriptor handle values and replace old file handles with new handles when I/O system calls are made. There are several possible implementation strategies for system-call interdiction:

Kernel-mode:

1. A Patched kernel
    1. Provide a source patch containing the needed kernel hooks that end-users must apply to their source tree and then rebuild their kernel.

2. Custom modules (drivers) overriding the kernel system-call table
    1. Similar to a patched kernel (1), but done at runtime; however, recent security improvements limit this approach to only functioning on 32-bit, and older 64-bit, Linux distributions.

3. The Linux Security Modules (LSM) framework
    1. This strategy is used by premier Linux security enhancements/auditors such as SELinux and AppArmor
    2. Using this method can easily cause conflicts with above mentioned commonly

used tools/systems which use LSM.

3. LSM requires a high degree of developer confidence as implementation errors could easily compromise the entire system's security and stability
4. There is a high technical barrier to entry as LSM is a complex API designed to be very flexible

User-space:

1. Custom shim library linking: Taking advantage of the face most applications do not make their system-calls directly but instead call the stdlib C functions provided by glibc (a dynamically linked library), it is possible to replace glibc with a custom shim that injects the desired logic.
    1. Would require identifying all glic/libc library code paths concerning relevant resource descriptors
    2. Would only work for languages and applications using glic/libc and NOT those making their own system calls directly or via another similar language runtime library
    3. Brittle; would be difficult to maintain as language libraries are modified and due to glibc's large size.

2. Ptrace
    1. API originally created for debuggers, now also used by auditing tools like Linux's strace command
    2. Well established and supported API, but the Linux implementation is not POSIX standard, as it implements a superset of the standard's functionality. (BSD for example only supports a subset of Linux's Ptrace functionality)
    3. Use doubles the number of system calls made (every system call becomes two).

3. Systrace
    1. Designed to improve on PTRACE by filtering system-calls in kernel

2. Only subscribed-to system-calls suffer the x2 system-call penalty
3. Not included by default in all Linux distributions.

In addition to system-call interdiction, the other primary local concern is one of implementing the throttling mentioned in section 2.2, this is as being a requirement for guaranteeing state convergence during live-migration. Since the primary mutable state of a target process is its heap, a means of detecting dirty pages, those pages which have been modified since being captured, is essential.

The following strategies were considered:

1. Forking on every iteration; this strategy takes advantage of Linux's approach of using copy on write (COW) pages for children. That is having child processes share the same memory as their parent process until a write occurs, and then allocating a new page and cloning the original page to it before allowing the write operation to proceed.

2. Insisting users install some of the kernel patch sets that provide a special character device for tracking changes [14].

3. Using a patch-set that extends the madvise() system call, adding a new command called MADV_USERFAULT, which can be used to handle page faults in user-space [15].

4. Using a system-call to mprotect(PROT_NONE) to mark all pages as read-only at the onset of every state delta transfer iteration, and then handle the resulting SIGSEGV signals to:
    1. Examine supervisor book-keeping to determine if the page was originally writable an we made it read-only, if so:

2. Mark the page as dirty in supervisor book-keeping
3. Make the page writable and allow the application to continue

Critera	Forking (1)	Out-of-tree character device patchset (2)	Out-of-tree madvise/userfault patchset (3)	PROT_NONE/SIGSEV handler (4)
Detect Heap Delta	**X**	**X**	**X**	**X**
Page Write event handling			**X**	**X**
Work on stock Kernel	**X**			**X**
High performance	**X**	**X**	**X**	

Table 4: State mutation selection criteria and matching approaches

All approaches are workable for the purpose of detecting heap deltas between transfer iterations. Approaches 2 and 3 would require a non-standard kernel, which is something this project is attempting to avoid. While approaches 1 and 2 are adequate for the the purposes of detecting the heap delta, they do not allow the actual page write event to be handled. Handling the page write event will allow us to throttle target process write progress in situations where the state delta between the target process and the remote clone is diverging rather than converging. That is to say, if needed we can artificially reduce heap write throughput by introducing a artificial delay to the page fault handler, which will ensure the state transfer thread makes more progress and the migration does not exceed the bandwidth available for state transfer, guaranteeing state convergence.

While not as performant as the other approaches, option 4, the mprotect/SIGSEGV-handler method is appealing because it functions with a stock kernel and provides both adequate delta detection and page write event handling for use in ensuring convergence. The

performance overhead actually inherently slows process heap write performance and therefore is synergistic with the goal of state convergence. Any implementation of this method is dependent on the supervising state capture process being able to modify the virtual memory mapping of the target process and use system-call and signal interdiction to handle memory faults.

2.5 Remote Migration Concerns

Many of the concerns of the remote process that is supervising a migrated clone are similar to those of a local process supervising a clone restored from a checkpoint. For example, the need for system call interdiction for PID and file descriptor fix-up is present in both situations, there are some concerns unique to running on a new host.

To evaluate if the scope of this project should include support the re-homing of network traffic as processes migrate between hosts, we must consider the network layer presentation of the target process to other network hosts. In many simple network application use-cases could be useful in situations where application availability is not handled at the application layer by a service discovery mechanism such as etcd and application depend on presenting their hosts IP addresses to the work (often exposed via a static DNS record). To enable the simple use-case of an IP address for each application, it is necessary for special a priori configuration to associate multiple IP addresses with the target process host so that each process which we would want to migrate has its own IP address. The original host and target host would have to be configured with a virtual/OS-level switch with a virtual NIC/IP address specifically reserved for a single process.

ARP announcements are the low-level method by which an Ethernet device can make its peers aware of it re-homing; this process usually only results in tens of milliseconds of downtime [16]. So additionally, network switches need to have been configured to not ignore unsolicited ARP broadcasts. Should these pre-requisite be met then:

1. The virtual NIC/IP address can be cloned on the target host
2. Unsolicited ARP replies will be issued from the target host informing routers of the new location of the IP address.

One issue with this approach is that during ARP based IP address re-homing, a small number of in-flight packets may be lost. One way this could be addressed is to configure the migration supervisor process or a kernel-based firewall on the original host briefly re-forward traffic while the ARP broadcasts take place. The function of the ARP broadcasts is to associate the IP address for the application with the MAC address of the host of the newly cloned process rather than the host of the original target process.

Another concern is the local persistent storage of a running process. Unless the migration tool synchronizes open files on the target and destination host, any program may fail to migrate if the migration starts during IO operations. This could be handled by transferring any files for which the target process has open file descriptors to the remote host. However, even if this took place, or a process was not engaged in I/O during the migration time-frame, it is possible the program expected other local files to continue to exist for future access. One solution to this dilemma is to require the application to have a pre-defined manifest that identifies local file-system and block devices on which it is dependent. Another possible solution is to require that applications eligible for migration store any files on a shared mount point that is available to possible migration target hosts. Examples of this kind of shared storage could be an NFS, SMB, GFS/2 or GPFS mount exposed via networking.

Another special case of the local file issue is local dynamically linked libraries (.so files) an application may be dependent on. Fortunately this case is easier to handle since these files are mapped into the target processes memory space, we could choose to treat them as memory space only. This leaves room for a future optimization where the dynamically linked libraries that exist on both the source and destination could be skipped during state transfer and simply mmap'ed on the target host side. There is still the pitfall of dynamically

linked libraries that are loaded late (after startup) possibly not being available after a process has been migrated to a new execution context; this can be addressed by the manifest approach or by requiring such libraries be stored on shared storage.

The last remaining challenge to remote migrations is state which is not stored in the application itself but rather within a cooperating process. A complex example of this which would make migration impossible would be applications which interface with the X11 windowing system. For this state to be eligible for migration special effort would be needed for the state capture supervisor process to interact with the X11 server directly. Another simpler example of this problem would be a process that communicates with a local MySQL server over the localhost IP address (127.0.0.1) or a UNIX socket. The cloned process would attempt to communicate with MySQL running on the migration target host and could fail or end up communicating with a different database instance (which may or may not be problematic). This problem could be solved by configuring the application to use a LAN IP address available to both the source and destination hosts. One final special case of the cooperating process state issue is when source and destination host are running substantially different versions of the Linux kernel. While a state capture and migration could be built to try to detect this condition, the availability of similar enough host kernels is a migration constraint that may be best left to the user to ensure.

3.0 Requirements and Analysis

3.1 Usability Requirements

While VM check-pointing, often referred as snapshotting, is very popular amongst users of virtual machines, despite similar snapshotting and migration software exsiting for processes, the tools that are process oriented are not widely used.

In my review of existing process state capture solutions, I found that existing solutions have one or more of the following problems:

1. Require the compilation of a custom-patched kernel
2. Alternatively from (1), require installation of a custom kernel module that exists outside of the primary Linux kernel source tree
3. Are not actively maintained and/or additionally often suffer from a poorly documented code base and a lack of man pages
4. Was broken by Linux kernel ABI and security changes, including a major change around 2004 when the Linux system-call table was marked as read-only by the linker at kernel compile time
5. Are designed to be used within the context of some larger framework such as a scientific/distributed computing platform.
6. Are not composable with other standard Linux UNIX-like tooling by being part of pipe chains with other utilities such as compression (gzip, bizp) and IO/network (dd, netcat) tools and supporting stdin/stdout
7. Are not available within the package management systems of mainstream Linux distributions, often requiring that they be downloaded and compiled by end users.

In an attempt to create a more usable implementation of process capture software the following objectives where made:

1. No kernel space code or modules should be required, the application should exist as a user-space program solely.
2. After project completion the source code, complete with code documentation, user documentation and compiled binaries, including Debian and Redhat packages, should be made available on a website or github. External contributions should be explicitly solicited.
3. The user-executable components should be composable with other UNIX-like tools and be usable as a foundation on which to build more advanced tooling, such as user friendly scripts and user interfaces.

3.2 Security Requirements

In addition to the usability concerns around custom kernel patching and kernel builds, the installation of custom kernel modules is also major security concerns. Historically, the easiest methods of injecting a root kit into a Linux system has been via these same mechanisms. Because of this many corporations do not allow application teams to use custom kernels and discourage use of custom kernel modules; therefore, relying on such a module would likely lead to a large number of users being unable to use the tool due to constraints imposed by their organization's operational security policies. Binary kernel modules are also particularly problematic, becuase unless cryptographic hashes for sources or binaries are provided users have little assurance that even if the original developer is trustworthy, that a module has not been tainted by an intermediary attacker. Therefore in addition to the usability concerns surrounding custom kernels and modules in Section 3.1, the security concerns also strongly suggest that the approaches using these technologies will not be acceptable to many end-users.

Another security concern is the high-level of privilege required for most state capture and migration programs to function. In this case, state capture mechanisms using kernel-space approaches have an advantage, as the already privileged kernel-space code can perform the

state capture when requested to do so by the user-space state capture supervisor process. This, however, may still be problematic as the increased user-space/kernel-space surface area presented for this coordination is likely to be less mature than stock kernel-user-space mechanisms, and of great interest to potential attackers.

State capture methods that can execute solely from user-space usually require root privileges to attach to the target process; however, in cases such as a PTRACE-based scheme, it is possible for a savvy user to avoid this unnecessarily escalation. This is done by explicitly allowing the state capture supervisor program only the necessary permission to attach to non-child processes. Example Linux shell code:

```
sudo apt-get install libcap2-bin # Install capabilities tooling
sudo setcap cap_sys_ptrace=eip /usr/bin/state-capture-tool #Give specific
privilege
state-capture-tool -targ-pid=1236 ... # Tool no longer requires root
escalation
```

Figure 2: Example of using setpcap to grant specific PTRACE privileges

The last reaming security concern is to discuss is the confidentiality of the resting (serialized) format and the transport of state of captured processes. It is reasonable to assume that many users will have sensitive data within their application heaps. For example, users bound by HIPPA, PCI and PIIA, and users protecting trade secrets, all need assurance that their confidential information is never transferred or stored in plain-text. To fulfill these use-cases introduces a security requirement that there must be an option to encrypt captured process state that is at rest or in flight, and to decrypt the same on the restore side.

3.3 Use-case Analysis

Per the discussion in Section 2.5, if fault-tolerance considered as out of scope for this project there are four remaining fundamental use cases:

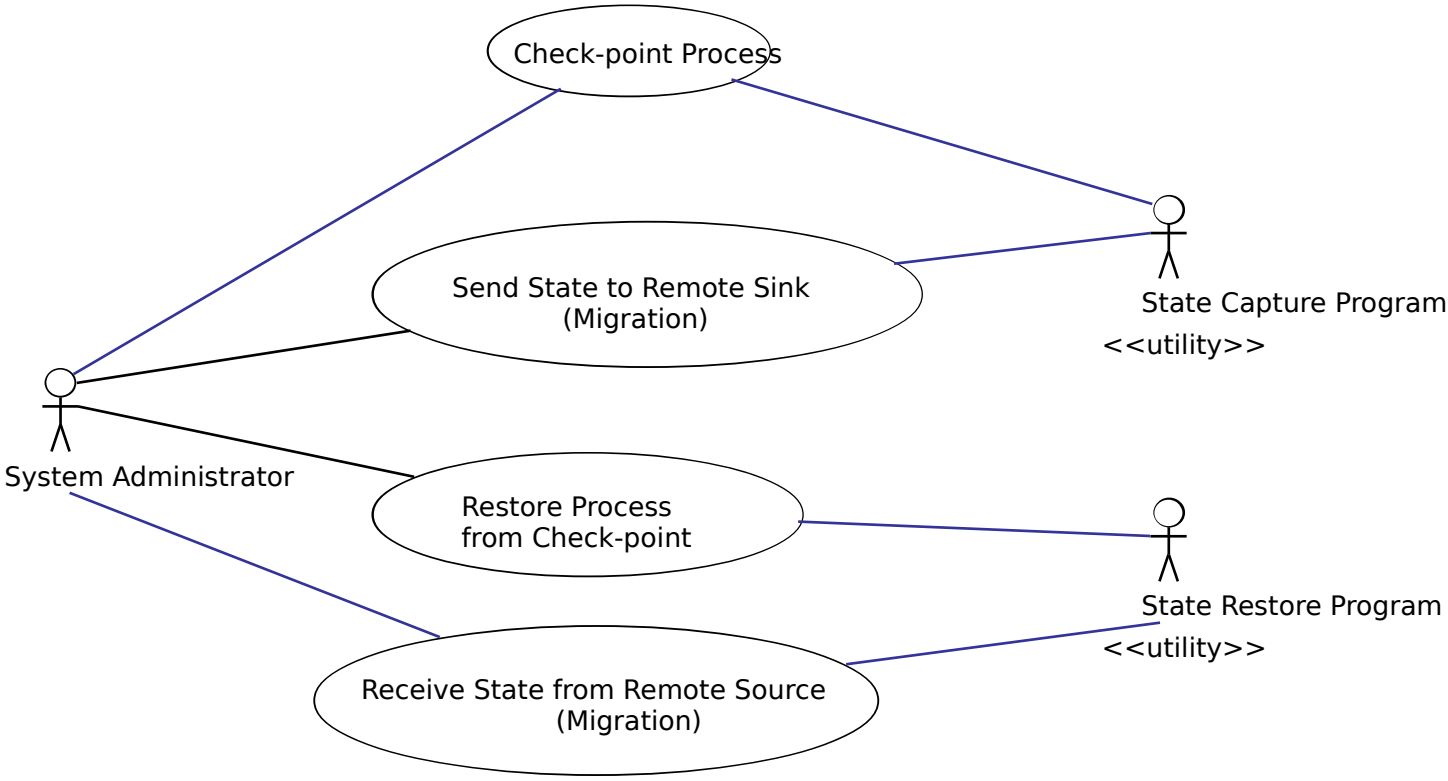

Figure 1: Core user-utility use-cases

1. Capture target state and create a persistent check-point snapshot of a process without harming the target process.
2. Perform the same kind of capture and serialization as in (1); however, rather than the destination being a local file, the destination will be a remote partner process that is receiving the transmitted state.
3. The reciprocal of (1); create a process from a check-point snapshot containing the

serialized process state.

4. The reciprocal of (2); listen for state from a remote sender and use that state to create a new local clone.

As described above the migration operation is actually equivalent to a clone operation, except in that the operation terminates the original process before resuming execution of the clone . To allow the user to actually migrate a process, use case (2) and (4) need a slight modification to transmit a halt flag. This flags the state capture program to listen to the remote partner for a signal of readiness, after which the original process will need to be halted. At this point the remote partner will start the the clone and upon success will signal the original state sender who will then terminate the original target process.

3.4 Project Scoping/Phasing and Design Assumptions/Limitations

The core use cases of this project can be implemented with significant variations in implementation complexity. For example, halted process versus live-copy migration differ greatly in developer effort, but for typical programs over a local area network offer similar usability to users. When defining the scope this project's implementation it was important to design the application in a way that will allow for incremental addition of more complex features without a great deal of refactoring and reorganization.

The following design and implementation assertions were identified:

Linux only: Due to low level and system-specific details of process capture and restoration, Linux, being the most widely used server operating system, was chosen. Additionally, the kind of analysis and integration needed would have been difficult to achieve with a proprietary system, such as Windows, that have a closed source base.

User space applications only: Migration of applications which directly interact with the

system
in kernel-mode outside of standard system calls are not supported. Most programs do not do this; however, this restriction implies the migration daemon cannot migrate itself. This also means software such as the X server could not be migrated.

Shared memory: Initially programs utilizing shared memory will not be supported; however, in the future it may be possible to support the live-migration of shared memory, so long as all processes engaging in the memory shared are part of the process tree that is migrating.

Shared filesystem: The filesystem the migrating application accesses must reside exclusively on shared storage. For example this could be a NFS or CIFS mount point or a mounted iSCSI /FcoE/AoE LUN used for a shared file-system such as GFS/2 or GPFS.

File-system containment mechanism: Limiting application access to the shared filesystem is left as a task for the user and other systems. For simple applications convention and configuration may be sufficient; for other systems many tool candidates for this policing include: chroots, OpenVZ, crun or docker.

External state prohibited: Dependence on state outside of the process tree being migrated will result in failures on the destination system. For example, because of assumption #1, an X11 instance cannot be part of a process tree that is migrating; therefore, an application that is using an X11 server in a stateful way could not be live-migrated.

Halt-and-Migrate: Non-live migration is sufficient for a majority of common use cases. Halted-process migration times for applications are primarily limited by network bandwidth, so an application with 300 MB of heap and a 50% compression rate would be relocatable between hosts connected by commodity 1 Gbit Ethernet in 1.2 seconds. Users who intend on migrating applications with very large heaps also tend to be leveraging enterprise class 10 or 40 Gbit Ethernet. Making similar assumptions, an enterprise user leveraging 40 Gbit

Ethernet would be able to migrate an application with 30 GB of heap in approximately 3 seconds.

Goals deferred to long-term implementation: For reasons of complexity management, the initial implementation phase was decided to be limited to migrating single-threaded applications only; this feature that would be a good target for inclusion on the next spiral iteration. Likewise the re-homing of network traffic was determined to be a goal best left for a future release iteration, as it requires a substantial amount of real-world lab hardware testing to gain confidence that the re-ARP MAC address announcements for relocated IP addresses are being properly interpreted by most network hardware in a timely manner.

Project Feature Road-map		
Goal	Phase (iteration)	Complete
Ability to capture snapshot	I	X
Ability to restore snapshot	I	X
Ability to transmit snapshot	II	X
Ability to receive snapshot	II	X
Ability to compress snapshots and transfers	III	X
Ability to encrypt snapshots and transfers	III	X
Ability to intercept PID related system-calls	III	X
Ability to intercept file descriptor system-calls	III	X
Ability to capture processes with multiple-threads	IV	WIP
Ability to capture process trees	IV	WIP
Ability to re-home network traffic (ARP / Socket migration)	V	Future
Ability to use systrace in preference to ptrace when installed	V	Future
Integration with Linux containers	VI	Future
Ability to enable live-migration	VII	Future
Support for similar operating systems: BSD, Darwin	VIII	Future
Ability to enable fault-tolerance	VIII	Future

Table 5: Project feature Road-map

4.0 Project Design

4.1 Separation of Concerns amongst Discrete Executable Components

The four fundamental uses cases identified in Section 3.3 decompose into two types of operations: those that capture process state and serialize it and those that take serialized state as input and construct a running process. This distinction lends itself toward implementing two distinct commands:

1. *pfrez* (process freeze): Capture the state of a process
2. *pthaw* (process thaw): Restore a process to execute from saved state

Originally there was consideration towards two additional commands:

3. *psend* (process send): Send a process to another host
4. *precv* (process receive): receive and execute a process from another host

However, it became apparent that pfreeze would always need some abstract sink to store its captured and serialized state, whether that sink be a snapshot file or a remote cooperating instance of pthraw. Likewise precv would be redundant as pthraw always needs a source to provide the serialized process state from which it creates a new process clone. The following is the command-line user help output of pfrez and pthaw:

Usage of pfrez:

```
   -compress string
      Compression mode: none | gzip | flate | snappy (default "none")
  -debug
      Debug: true | false, if enabled outgoing data will be displayed
  -dest string
      Output sink: stdout | tcp|udp:host:port | unix:socketpath |
snapshot-filepath (default "stdout")
```

```
  -dial-timeout duration
      Optional: Duration to wait for socket level connection to be
established
  -encrypt string
      Encryption mode: none | AES-CFB|AES-CTR|AES-OFB:keypath (default
"none")
  -halt
      Halt the target process after state capture and transmission is
complete
  -pid int
      PID of process to be frozen (default -1)
  -write-timeout duration
      Optional: Duration to wait transmitting data to an active stream
before timing out-compress string
      Compression mode: none | gzip | flate | snappy (default "none")
  -debug
      Debug: true | false, if enabled incoming data will be displayed
  -dest string
      Output sink: stdout | tcp|udp:host:port | unix:socketpath |
snapshot-filepath (default "stdout")
  -encrypt string
      Encryption mode: none | AES-CFB|AES-CTR|AES-OFB:keypath (default
"none")
  -pid int
      PID of process to be frozen (default -1)
```

Usage of pthaw:

```
    -debug
      Debug: true | false, if enabled incomming data will be displayed
  -keydir string
      Optional: Directory containing decryption keys
  -loader string
      Optional: Alternate path to loader executable
  -read-timeout duration
```

```
      Optional: Duration to wait for incomming data on an active stream
before timing out
  -src string
      Input source: stdin | tcp|udp:port | unix:socketpath | snapshot-
filepath (default "stdin")
```

See Section A, the appendix, for examples of invocation.

While at first it may appear the many encoding (and implied decoding) options above would be difficult to manage from a complexity standpoint, in practice all of the encoding options are composable and can be layered as needed to fulfill the user-specified flags. This is possible because all streams implement a common io.Reader/io.Writer interfaces, and so long as the constructor for each wrapping layer takes another io.Reader/io.Writer as an input argument, composition is trivial.

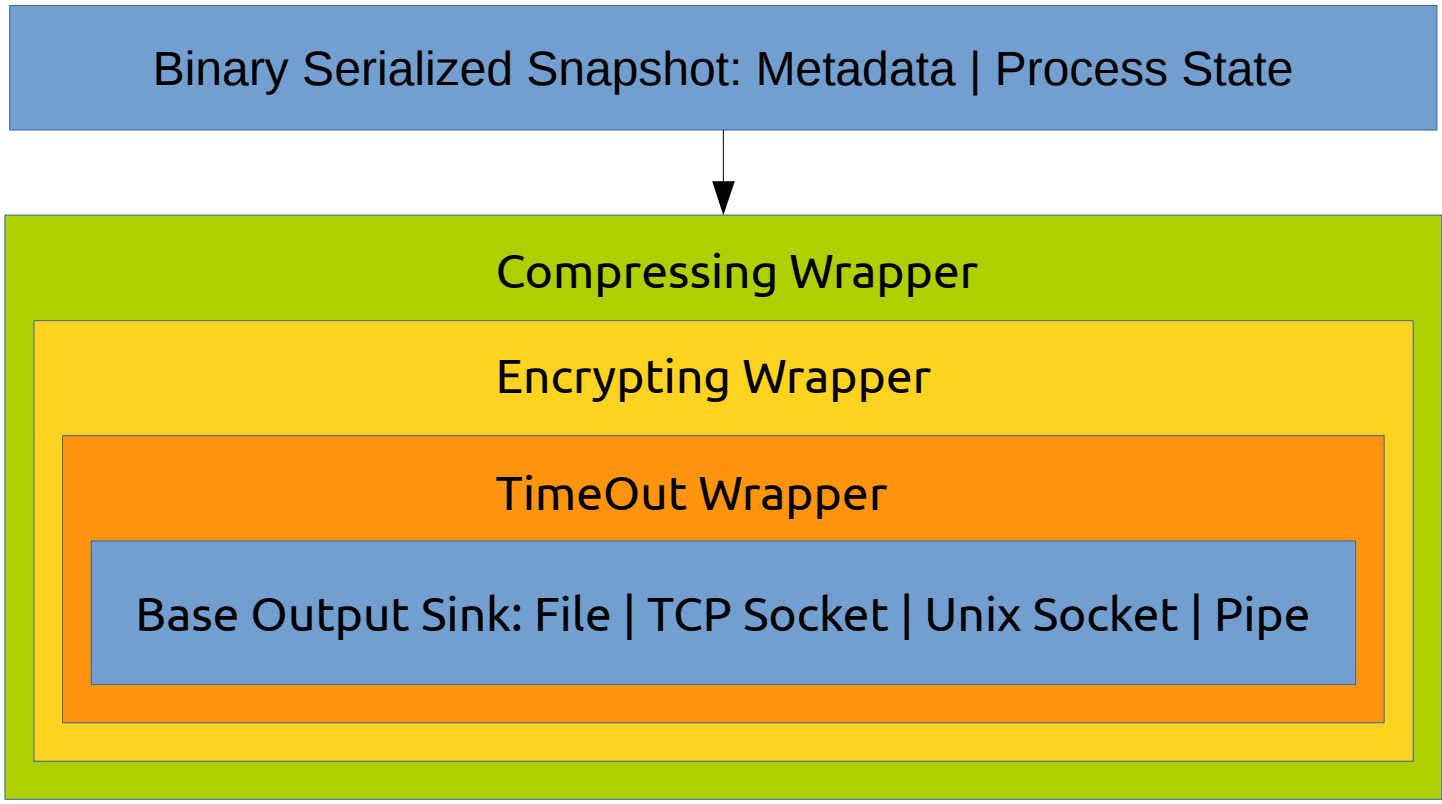

Figure 3: Composition of all input/output stream wrappers available

Before the serialized output stream is sent to the selected output sink, it is prefixed with a plain-text header that states the length of the prefixed header. This allows it to be delineated when read from the data stream that follows it, as well as provide the necessary information for the restore program to apply corresponding stream unwrappers. This includes information such as the compression and encryption algorithms applied, if any, as well as any other information needed, such as AES pre-shared key-names and initialization vectors.

4.2 Separation of Concerns amongst Logical Programmatic Components

While the functionality of the application is divided between two executables, which exception of the "main" packages for pfrez and pthaw, the remainder of the code base is organized as a number of shared packages each with their own unit tests. The goal was to maintain separate command-line applications for the benefit of the user, rather than one monolithic executable with a confusing combination of mutually exclusive command-line flags, but have these executables share a common code base. The packages used in this project are arranged as follows:

- lib: This parent contains the code that is share across pfrez and pthaw
    - *errs*: A package to extend the stdlib assist with generic error handling
    - *pmaps*: A package for representing process virtual memory maps
    - *ptrace*: A Go wrapper on the PTRACE kernel API
    - *preader*: A package that contains classes which read process state from an executing process or a process snapshot
    - *pwriter*: A package that contains classes to write process state to a new process, to a snapshot or to the console as debugging information

    - *psupervisor*: A package that provides classes which supervise the execution of a cloned process performing tasks such as system-call interdiction

- *pfrez*: The pfrez main() was well as encryption and compression logic
- *pthaw*: The pthaw main() was well as decryption and decompression logic
    - *pload*: A shim-program that, unlike the rest of the Go source code, is written in C and is used for creating the process that will eventually be transformed into the cloned process. It communicates with pthraw via pipes it is provided at startup.

The entity relationship can be seen on the following diagram (Go stdlib classes omitted):

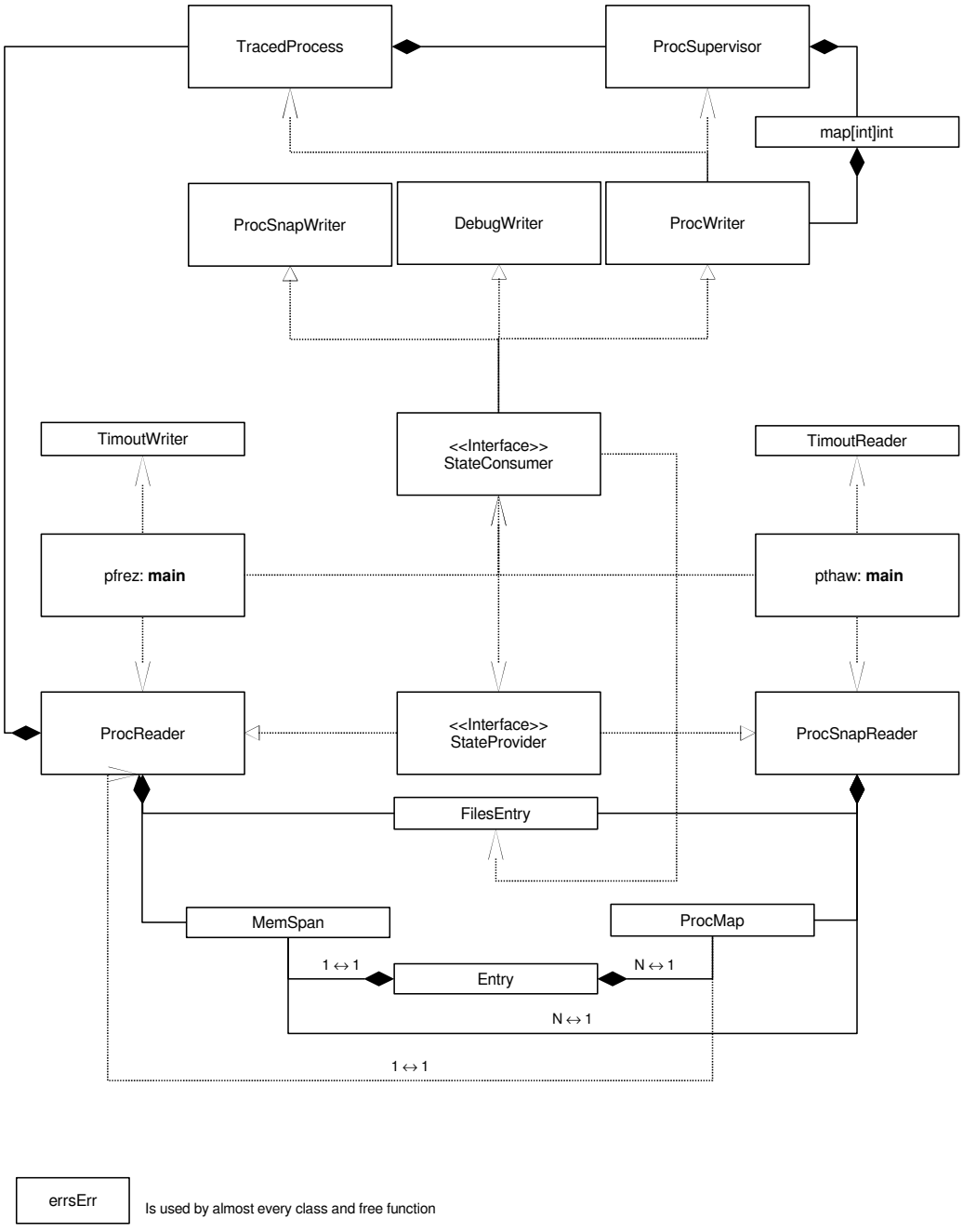

Figure 4: High Level UML Diagram

4.3 Discussion of Cohesion and Coupling

As is the case with any large software project, there was a desire to maximize maintainability by minimizing coupling and maximizing cohesion to ensure development velocity remains high. To this end the cohesion and coupling was evaluated at the package, file and class/struct levels. *Exporting* the term Go uses to denote that a field or method is visible in the context of other packages. In this discussion I will refer to Go structs as classes since that term is more familiar to readers; Go structs may have member functions attached and support both exported and non-exported fields and methods.

Coupling was minimized by creating a few core interfaces that were repeatedly referenced, and cohesion was sought by grouping implementations of these interfaces in a package created for that purpose. For example ProcReader (process reader), and ProcSnapReader (process snapshot reader) are placed together in the preader (process reader) package.

Besides implementations of interfaces, the only other logic present in these packages is unit tests, which gives unit tests access to unexported implementation details. Similarly for the sake of reducing coupling, integration tests which test the integrations between packages, were placed outside of the packages they test the interoperability of; this limits their access to the exported public signature of the packages involved.

All packages contain either a single purpose class or multiple classes which implement the same interface, this results in most coupling taking the form of external coupling. Care was taken to ensure the export signature of each package was minimal, which helps act as a natural safe-guard and eliminates content coupling and common coupling.

Observe in the following table the environmental coupling present (excluding Go standard library packages and unit tests). The equation for environmental coupling is

$Coupling(C) = \frac{1}{w+r}$; therefore, the lower the value, the more adventurous the design scenario becomes [17].

Package Name	Packages Imported (*w*)	Packages Importing (*r*)	Env. Couplin
main (pfreeze)	7	0	0.14
main (pthaw)	6	0	0.17
lib	2	3	0.20
errs	0	10	0.10
iotimeout	1	2	0.33
pfiles	1	3	0.25
pmaps	1	2	0.33
preader	5	2	0.14
psupervisor	2	1	0.33
ptrace	1	3	0.25
pwriter	5	2	0.14
transpenc	1	2	0.33
Average	2.67	2.50	0.23

Table 6: Project package fan-in/fan-out and environmental coupling calculations

With the exception of the sentinel values used for error objects, no global variables are instantiated outside of the main packages; therefore no exported globals declared meaning the project has virtually no global coupling. This is both good design and a natural outcome of the incompatibility of globally shared state and unit testing.

5.0 Implementation

5.1 Development Life-cycle

Due to the great number of unknowns regarding requirements and technologies a traditional waterfall model was unsuited to this project. The project began with an extensive literature review, the objective of this review was to determine the requirements of likely users (see Section 3.1), avoid potential security concerns (see Section 3.2) and gain an understanding of the underlying technologies most suitable for use in implementation (see Section 2.4). For these reasons, once the literature review was completed a development very similar to the hybrid waterfall approach known as the *spiral model* was used.

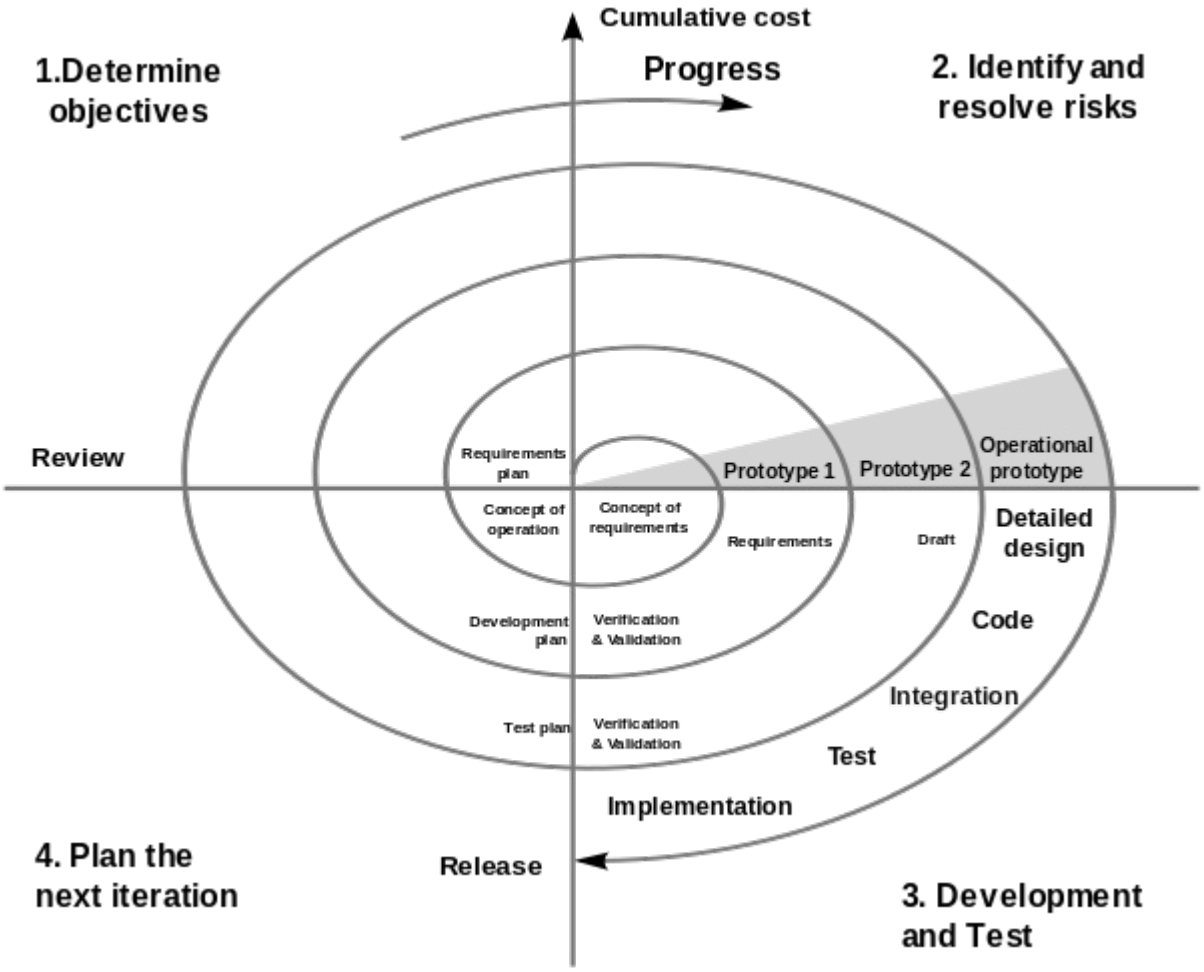

Figure 5: Graphical representation of the Spiral Model of software development [18]

This approach allowed frequent prototypes that encompassed the requirements as currently understood to be developed and evaluated. Very frequently spiral iterations were abandoned, and work was reset to the previous iteration's code-base when an incorrect design decision or assumption was made. For example, at one point all the functionality was built into one monolithic executable with a confusing combination of mutually exclusive command-line flags; or in another case unsuitable technologies were mistakingly used when they had been rendered obsolete by changes to the Linux kernel. The downside of this approach was that there was a large quantity of code written that was never used; however, this approach is still commendable for any project with a high degree of uncertainly where experimentation is

required. The work-flow of resetting to the previous iteration as a starting point is made simple by using source control software.

The only major exception to the above mentioned spiral model methodology look place during a complete code rewrite to enact a language port from C++ to Go (See Section 5.3); at the onset of undertaking this rewrite, there was a waterfall like high-level redesign that wwas synergistic with the code refactoring required to enact the language port.

5.1 Selected Technical Approaches

The first major technical decision to be made before implementation could begin was which low-level methods should be used to interact with the target process and capture its state. At the outset of this project, an assumption was incorrectly made that interdicting system-calls would be straightforward and could be accomplished by overriding the system-call table with custom hooks. It was subsequently realized during the research and discovery phase that, while this had been a popular approach in the past, this method would not be viable because recent Linux kernel linking changes had rendered the system-call table immutable. This was done in an effort to harden the kernel and hamper the installation of root kits. Furthermore, this method would have required writing a custom kernel module in C that the user-space portion of the program would communicate with. The reliance on a third-party kernel module and the expansion of the user-space/kernel-space interface would have been unpalatable to many users from a security prospective (See Section 3.2)

Due to these concerns the decision was made to use the PTRACE API, specifically the PTRACE API as presented by the Linux kernel, which is a superset of the POSIX PTRACE API that other systems such as BSD implement. The PTRACE interface is arguably inferior to the newer systrace API that was designed to replace it; using PTRACE as opposed to systrace involves twice as many context switches for example. However, systrace is not universally enabled in kernel builds yet, and it was deemed more import at to support a wider range of users at initial implementation time. Inclusion of systrace as an optional

method of state capture and system-call interdiction to be used when locally available is slated for a future implementation iteration (See Section 3.4)

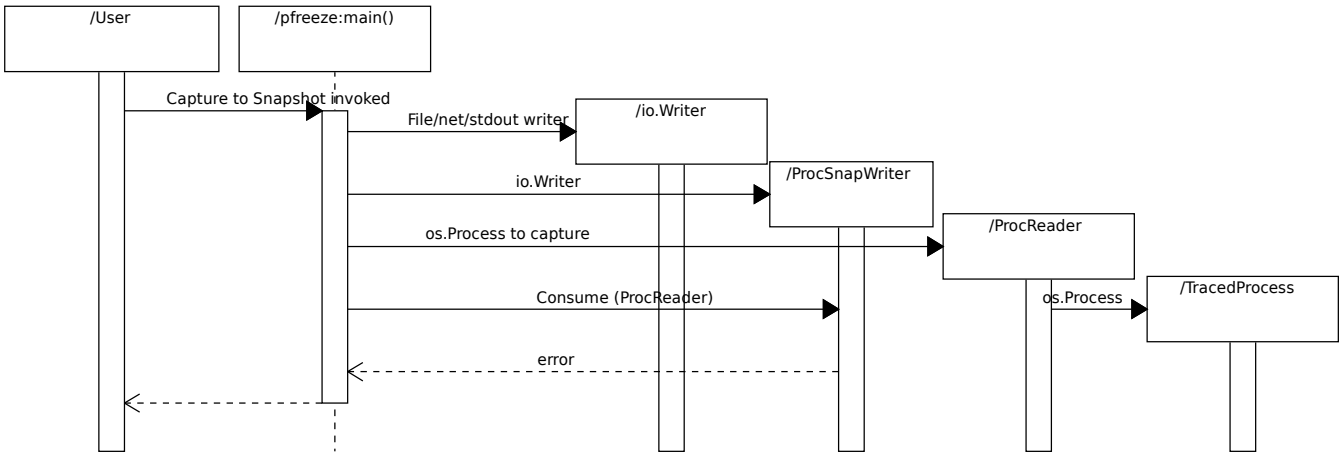

Figure 6: Sequence diagram for process state capture

5.2 Technical Challenges

The other major technical challenge was how to take the serialized state of the captured process and restore it to a running process. While this is similar to the executable loading work that happens every time a user executes a new binary; however, the exiting functionally does not concern itself with maintaining the state changes of processes and thus existing OS APIs for program loading facilitates were of limited usefulness.

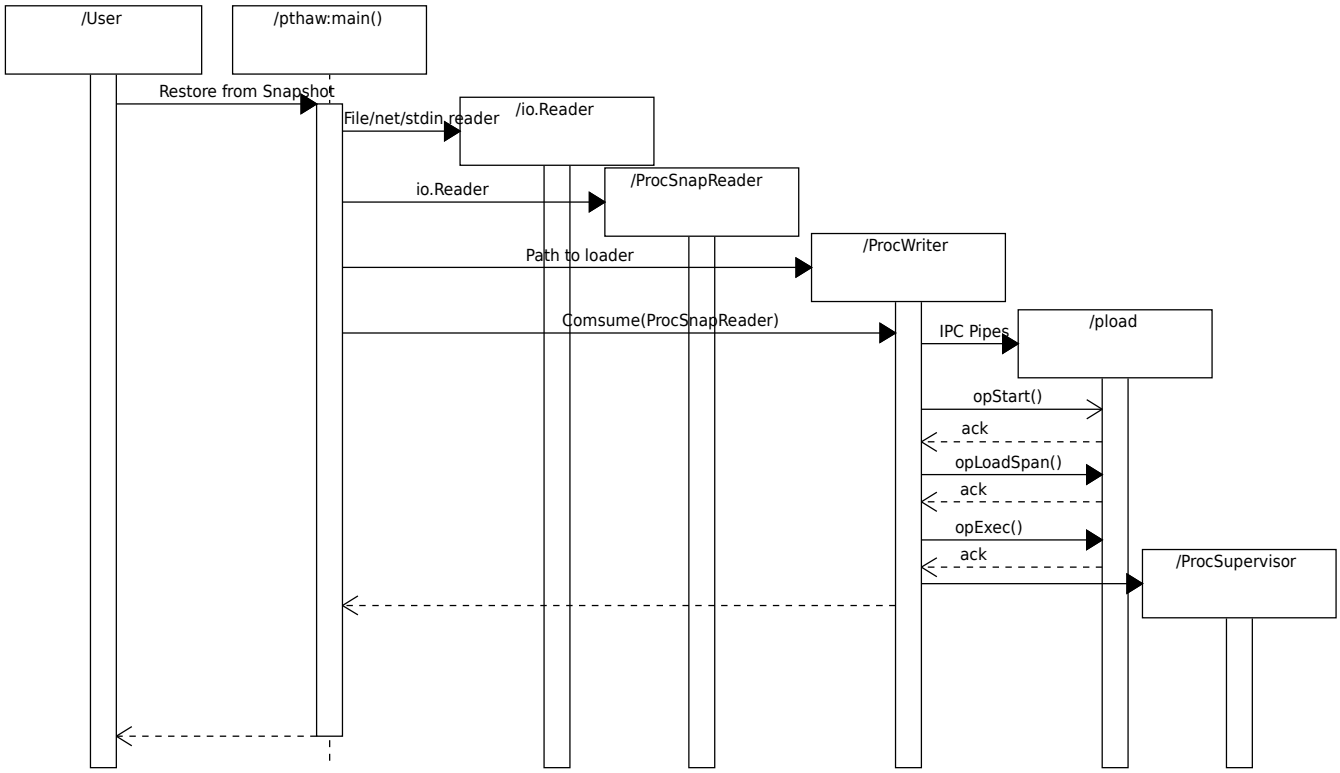

Figure 7: Sequence diagram for process state restore

The approach taken for the state restoration process was to have the pthaw, the program supervising the restoration/cloning process, to create a child process. This child process is pload, a small C program. When pthaw invokes pload it connects a pload file descriptor to a pipe that it has created for its child. Pthaw can then proceed to read, decrypt, decompress, deserialize and sanity-check the inbound process state stream. Pthaw then sends a normalized form of this stream to pload via their shared pipe/file descriptor link. The communication between pload and pthaw is bidirectional, and pload communicates back to pfreeze the success or failure of each state change. To avoid interfering with the state that pload is populating, a custom linker script is used to place pload's own executable logic in portion of its address space that is rarely used in Linux applications. In the future, pload could support relocating as needed if address space conflict is determined to be an issue for users. Pload's need to have absolute control of its address space to guarantee it would not

interfere with the state loading process created a special challenge in that pload needed to be statically linked. This mean not only not loading the glib C stdlib dynamically linked library into its memory space, but also in an effort limit its absolute size it not does include the C standard library at all. This meant writing the pload program in such as way that it only relied on OS system-calls. The following is an example of the code needed to perform a simple and familiar C *puts* function call to write a message to stdout:

```
#define stdout 1

int64 puts(char* msg) {
	return fputs(msg, stdout);
}

int64 fputs(char* msg, int64 fd) {
	return write(fd, (void*)msg, strlen(msg));
}

int64 strlen(char* msg) {
	if(msg == 0) {
		return 0;
	}
	char* msgStart = msg;
	for(; *msg != 0; msg++);
	return msg-msgStart;
}

inline int64 write(int64 fd, void* buf, int64 len) {
	return syscall3(SYS_write, fd, (int64)buf, len);
}

inline int64 syscall3(int64 syscallNum, int64 arg0,
 int64 arg1, int64 arg2) {
	register int64 syscallNum_ __asm__("eax");
	register int64 arg0_ __asm__("edi");
	register int64 arg1_ __asm__("rsi");
	register int64 arg2_ __asm__("edx");
	syscallNum_ = syscallNum;
	arg0_ = arg0;
	arg1_ = arg1;
	arg2_ = arg2;
	asm volatile (
		"syscall"
		: "+r"(syscallNum_)
		: "r"(arg0_), "r"(arg1_), "r"(arg2_)
		: "%rcx", "%r11", "memory"
```

```
        );
        return syscallNum_;
}
```

Figure 8: C Code illustrating the complexity of “puts” without a C standard library

Once the complete process state has been processed by pthaw and transmitted to and installed by pload, pthaw can now start the restored process clone. This is accomplished by sending pload a final command that instructs it to jump to the address of the program counter as captured from the target process. Pthaw will then continue to operate in the background as a supervisor, using classes in the psupervisor package to monitor the cloned process and perform system-call interdiction to fixup child process system calls as needed (See Section 2.4 for system call interdiction details)

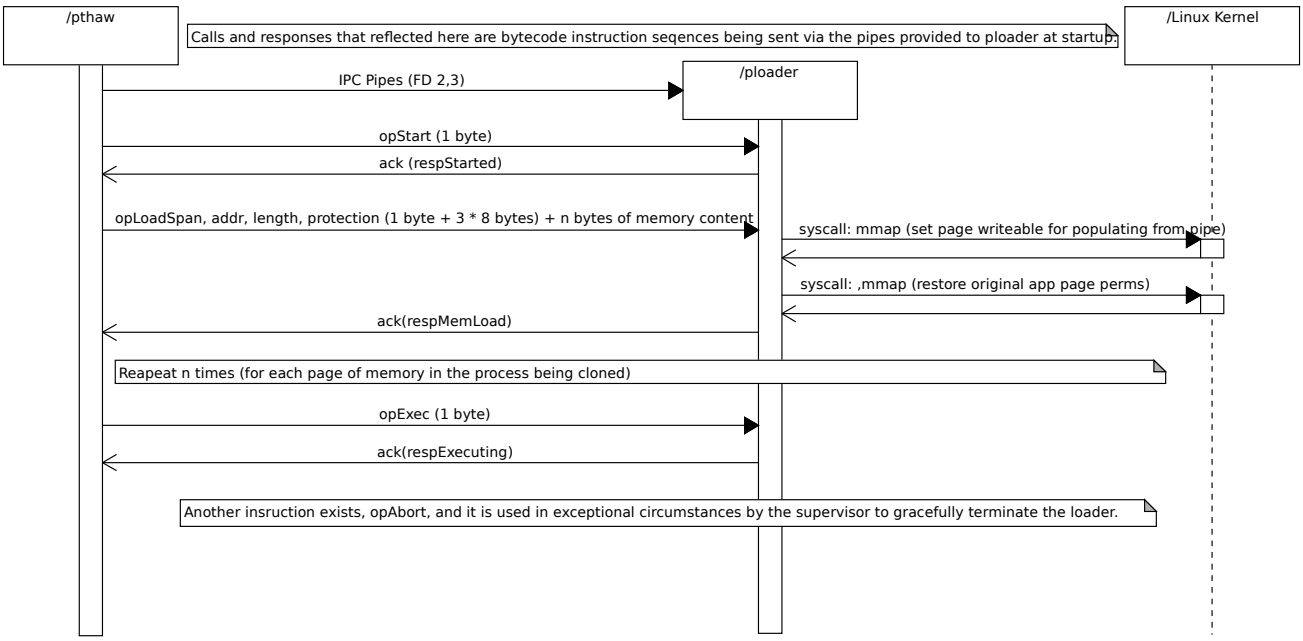

Figure 9: Detailed sequence diagram illustrating pthraw-ploader communication during state restore

Besides the need to write ploader without use of the C standard library, there were many other technical challenges along the way. As previously mentioned, the discovery that the

Linux kernel had made the system-call table read-only was a major early challenge. At one point it appeared system call interception was not happening, when in fact the application's glibc instance was caching the result of the system call the first time it was made and then always using that result in the future. This was unexpected; upon further investigation it was found that the maintainers of glibc (GNU) and the Linux kernel disagreed if this was proper behavior. Linus Torvalds in fact agreed that such behavior would break state management applications similar to this one and was an example of an optimization which appears worthwhile in benchmarks without having real world value [19].

Many of the problems encountered were not documented in the relevant source code or man pages. For example the Linux proc filesystem exposes the raw memory contents of applications to root and ptracing application with a file: /proc/<PID>/mem. While this file could be read as expected using the read system system call, the read-at system call intermittently failed. Once this realization was made, it was simple to work around this issue by calling seek, then read, for every record, but debugging this behavior was non-trivial.

Similarly, towards the end of development, there was a frustrating period where tests which normally passed would fail intermittently; this ended up being caused by an undocumented race condition created by the interaction between the Go runtime's threading model and the signals PTRACE uses to notify callers of system call interception. A single call to tell the Go runtime to use a single-thread for all system calls easily fixed the issue, but only after weeks of debugging.

5.3 Discussion of Language Technologies Utilized

The initial implementation of this project was written in C++. C++ was selected primarily due to being the only language that met the technical requirements of being highly interoperable with C (being able to call C APIs directly) and an easy language to interface with system Linux system calls which are written in C due to its C heritage, its ability to

forgo memory safety and provide very low level memory manipulation as well as the strong suggestion of the MSE program that the language used be object oriented.

During the course of this project, Google released a compiled multi-platform systems/services oriented programming language called Go. Two of the three core Go language designers were ex-Bell Labs team members Ken Thompson and Rob Pike (the third being Robert Griesemer, known for his work on Java Hotspot and the V8 Javascript engine); has led to Go sometimes being referred to as "C++ as brought to you by the people who made C" (rather than as created by Bjarne Stroustrup who actually extended C to C++). Go is built on a similar syntactic style to C (with some Pascal influences) with the addition of first class functions, closures, anonymous functions, structural typing (aka compile-time duck typing), first class CSP style concurrency primitives (such as thread-safe queues called channels), interfaces and interface inheritance. All of this functionality exists in a type-safe, memory safe and garbage collected language that is both compiled to native executable and portable (Go programs are write once, compile many times for each target).

Go is similar to C++ in that a programmer is free to write both structured and object-oriented code as the situation and their taste dictates, so for example, the main function in a Go program is a free function in the main package. Go differs from traditional object-oriented languages in that it does not allow inheritance amongst concrete types (only between interfaces), instead providing a special form of composition called embedding that allows for convenient code re-use without the complexity of an "is-a" relationship. Go is also notably different in that access modifiers apply only at the package level with types and fields being either "exported", visible to external package, or not.

One subtle difference between Go and many other modern languages is the ability of Go functions and methods to return multiple values. This is often first noticeable by its direct effect on Go's error handling strategy; it is standard practice for Go functions that may fail to return a result and an error (a built-in type), which is checked by the caller in lieu of the

exception handling provided by other languages. For example the following Java:

```
FileReader reader = null;
try {
    reader = new FileReader("someFile.ext");
    //Process reader...
} catch (IOException e) {
    throw MyReadOpFailedEx(e);
} finally {
    if(reader != null){
        try {
            reader.close();
        } catch (IOException e) {
            throw MyResourceCleanupFailedEx(e);
        }
    }
}
```

Would be represented in Go as:

```
reader, err := os.Open("someFile.ext") //Declare and instantiate
using types from function signature
if err != nil {
      return errors.New(`Failed to open: "someFile.ext"`)
}
defer reader.Close() //Deferred stmts run in FILO order at scope
collapse
```

Figure 10: Demonstration of how Go differs from C++/Java in respect to exception handling

While working on this project, the author was employed by a startup (a later a large corporation) engaged in system software development using Go. Once it became apparent that the author was much more productive in Go than C++ and it was realized that Go provided facilitates, called cgo, to easily invoke C code, as well was the ability to directly manipulate memory in using the “unsafe” standard library package, then rewriting the existing project in Go became attractive. The tipping point came when the author was spending more time debugging C++ memory management related crashes seen during the execution of integration testing scripts than adding new functionality to the project. Upon the realization that updating the project to use modern C++ auto-pointers and C++11 features

would take at least a comparable of time as a full port to Go, a Go port commenced.

The size of the work in progress code base was reduced by approximately sixty percent due to a combination of refactoring, a much more comprehensive Go standard library, increased language derived code density and Go's lack of header files. The benefits of working in a language that the developer is highly productive in became immediately obvious when the lines of code committed per week during active weeks increased dramatically after the port. Other significant benefits included unit testing support built-in to the default Go tool-chain without requiring third-party tools or libraries. Previous testing and been manual and scripted integration tests only. The presence of a much more comprehensive standard library in Go than C++ which reduced the number of custom utility functions and third-party libraries necessary (Like BOOST for C++ or Apache commons for Java). The elimination of custom utility functions in favor of standard library routines contributed to reducing the lines of code in the code base and improving reliability.

5.4 Summary of Development Environment

Development of the project was conducted on machines running various releases of Ubuntu Linux LTS (Long Term Support) running on the Intel implementation of the amd64 architecture.

Prior to the project rewrite from C++ to Go, development was conducted using Netbeans with a C++ plugin using GCC as the back-end compiler and SVN for source control. GDB was regularly used for debugging.

After the rewrite LiteIDE was used as the IDE using the standard golang.org Go compiler for the Go portions of the project and GNI gcc for the C portions (pload and test programs). Arguably due to the proliferation of unit tests and following typical test-driven Go development practices, a debugger was no longer used. The transition from SVN to git took place at the same time as the Go port and was also accompanied by a transition from SVN hosted by Assembla to git hosted on my personal server running gogs (an opensource git

web UI written in Go).

When testing host-to-host migrations manually, testing was accomplished by using a combination of transfers to localhost via loopback interface, by using virtual machines communicating over a virtual switch and by using physical machinea (my desktop and laptop) over the local area network. At the beginning of this project, VMware ESXi hosted on a dedicated machine was utilized; however, by the conclusion of the project, usability and performance enhancements in Oracle VirtualBox enabled that more convenient desktop solution to be utilized.

6. Testing

6.1 Overview of Testing Objectives

Driven by a desire combat the challenge of the brittleness cuaed by integrating with low level APIs such as PTRACE, to create a project that will attract an open-source community after public release, it was a goal from the onset of this project to have a suite of integration tests. When the project was later rewritten in the Go programming language, this goal was expanded to include unit testing as well.

6.2 Unit Testing

The unit tests for this project had 5 primary objectives:

1. Provide unit tests to detect defects within packages as changes are made.
2. Provide unit tests to test higher level packages that integrate multiple lower level packages.
3. Provide unit tests to act as a form of documentation to new developers and lower the barrier to becoming a contributor.
4. Provide specialized unit tests called 'unit benches' to allow before and after performance comparisons to detect performance regression for pull requests that modify packages.
5. Requiring unit tests for all packages has the beneficial side effect of ensuring all code is unit testable. Even without the presence of the unit tests themselves, the level of abstraction and separation of concerns required to create unit testable code tends to increase project maintainability.

Go's excellent built-in standard library support for unit testing made this endeavor much easier than other languages where third party frameworks must be learned, evaluated and integrated. The suite of unit tests has been configured to run as part of every build to better fulfill the roll of a smoke test. Go has a special execution mode that enables a data-race

detector; while enabling this tends to slow unit tests down, it drastically increases the likelihood of detecting data-races in multi-threaded code as the Go runtime will panic when a data-race is detected.

6.3 Integration Testing

While unit testing is very capable of testing low level code integration, unit testing alone is less capable of detecting failures between the discrete system level components that result from the build process and system specific issues at install time.

A combination of bash and expect scripts was used to create some simple test cases that work by calling pfrez and pthaw in the same manner that end users might. This acts as a final level of integration for developer testing and can even be executed post-application install to verify end-user systems are operational.

When this project is released as a public open source project, the intent is that these scripts will be called by the Debian and RedHat post-install event hook for each respective system's packages to validate that the software will function correctly on the end-user system.

6.4 Continuous Integration

One of the additional advantages of having a suite of unit and integration testing is triggering these tests to execute on code submission events. Proof-of-concept testing has been completed where unit tests and integration tests were triggering in response to the creation of git (source control) pull requests. This kind of integration will streamline the code review process as the project moves from a solo effort to become a community-maintained project, because maintainers and contributors will have instant feedback in regards to unit and integration test success of pending pull requests.

7. Conclusion

7.1 Challenges

This project had some unique challenges due to its requirements-gathering phase being less straight forward than a more typical forms-over-data type application. The first step was to conduct a survey of existing solutions and attempt to make determinations in regards to why the solution was not popular, and thus failing user acceptance, or why the project had apparently failed and was no longer being maintained. After this survey, an extensive literature review needed to be conducted for both the theoretical methodology and the platform-based technical options that shaped the project objectives, goals, scoping, assumptions and limitations. Without a complete understanding related methodologies, such as migration verses live-migration versus fault-tolerance it would have been impossible to determine the proper scope for this implementation.

All traditional user-space applications are dependent on their operating systems kernel. But this project's third party dependency exhibited far more unavoidable uncoupling as the application needed to use many different OS interfaces such as the proc file-system and PTRACE. Both of these facilitates were not originally intended to be used in conjunction with one another and often revealed unstable or undocumented kernel specifics. Kernel design changes, bugs and undocumented interactions between technologies often made for difficult to resolve bugs and unexpected regressions during updates. This brittleness was addressed by way of extensive unit and integration testing as well as by minimizing surface area; this has been mostly successful in managing this external coupling challenge moving forward and at the very least makes environmentally caused regressions immediately evident.

Another challenge was the fact that the language which seemed best for implementation at onset, C++, was notably lacking (needed many 3rd party libraries, not memory-safe, no first-class support for unit testing) and had to be replaced later by Go to improve productivity and

maintainability. While this transition was worthwhile overall, it represented a large investment of effort and was not without its own trade-offs. One example is documentation tooling; Java and C++ have fantastic support for representing their design as UML in terms of notation, tutorials, code-to-UML and UML-to-code generation. Go has language features such as multiple return values, and CSP derived concurrency primitives. such as goroutines (coroutines multiplexed over multiple threads) and channels (thread-safe queues used by goroutines), that can be challenging to represent in many UML tools. Go is both a much newer language and has come into existence during a time when many developers are eschewing some more traditional forms of documentation in the name of agility, and so lacks much of the UML tooling and required much more manual documentation work.

7.2 Project State, Future and Continuing Work

At this time the software implemented in this project can fulfill all the basic process capture, restore and migration use cases. Processes can be snapshotted for future restoration or they can be paused, cloned and migrated. Currently only processes that limit their activity to local I/O and single threads are supported.

The future of this project lies in two area: The first area is technical enhancements that can be layerd on the flexible architecture that has been built to support more complex process use-cases such as multi-threaded applications, entire process trees (cooperating processes) and processes engaging in network I/O that will need Ethernet/IP re-homing (See the project road map Section 3.4).

The other area of project growth resolves around turning this one-man, academically-driven project into an open-source, community driven project. The initial steps are straightforward, such as choosing an IP license and placing the source code in a publicly accessible repository on a site such as github or bitbucket. This would be folllowed by setting up continuous integration so contributors get automated feedback on their submissions. But the hardest part will be building a community of users, contributors and fostering healthy community culture.

Another future opportunity that is both technical and community in nature is engaging with the various Linux container projects to see if pfreeze and pthraw could be a useful component of their solution stack. Linux containers provide an easy way to ensure that processes being migrated do not violate any migration constraints and protect users from having failed migrations when they unknowingly violate a state capture/restore limitation.

Concurrent with this project's implementation was the development of another similar project called CRIU [20]. While CRIU (Capture-Restore-In-Userspace) is advertised as being “mostly” in user-space, in fact the project ended up submitting and maintaining many patches to the Linux kernel to accomplish their objectives. Originally the Linux kernel developers were skeptical that CRIU would be a successful project and required that related patches contain identifiers to allow them to easily be removed from the Linux kernel [21]. This project did not even attempt such patches, due to manpower limitations and making the assumption that the Linux kernel team would not have accepted them and any patches would have to be maintained in an out-of-tree patch-set which would be difficult to maintain. CRIU has a much larger development team than this project, allowing them to engage in kernel maintenance and progress farther along, supporting features such as network re-homing. However at the same time, CRIU is in some ways is more limited in the assumptions it makes about migrated processes, such as their tty/pty connections. CRIU's use of kernel integrations means while pfrez/pthaw could easily support other operating systems, such as BSD/Darwin in the future, CRIU is much less portable. CRIU is extremely interesting as an alternative implementation of state capture-restore and will be useful for sanity-checking both the design and user experience of this project. Ultimately solution space competition is healthy and the solution presented here has several commendable properties, such as being written in the same language as the popular containerization platform Docker (Go), and more rigorously conforming the the UNIX utility design patterns that allow for composability in piping and redirection.

In conclusion pfreeze and pthaw have an extensible architecture, a solid testing foundation and a clear technical and community road map; they should be an increasingly useful utilities to the Linux and Linux container community going forward.

8.0 Bibliography

[1]: IMEX Research, NexGen Data Centers: The Rise of Virtualization, 2007, http://www.imexresearch.com/newsletters/-jan_07b_new.htm

[2]: Erich Amrehn, Jim Elliott, zVM01: 45 Years of Mainframe Virtualization: CP067/CMS and VM/370 to z/VM, 2012

[3]: Virtual Machine, 2016, https://en.wikipedia.org/wiki/Virtual_machine#Process_virtual_machines

[4]: Gerald J. Popek, Robert P. Goldberg, Formal Requirements for Virtualizable Third Generation Architecture, 1974

[5]: VMware, Inc, Security of the VMware vSphere®Hypervisor, 2014, https://www.vmware.com/files/pdf/techpaper/vmw-wp-secrty-vsphr-hyprvsr-uslet-101.pdf

[6]: Fei Xu, Fangming Liu, Hai Jin, Vasilakos Athanasios V., Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions, 2014

[7]: , Exponential Docker usage shows container popularity, 2016, http://www.businesscloudnews.com/2016/02/11/exponential-docker-usage-shows-container-popularity/

[8]: Jonathan Corbet, Process containers, 2007, https://lwn.net/Articles/236038/

[9]: Usage of operating systems for websites, 2016, https://w3techs.com/technologies/overview/operating_system/all

[10]: Usage statistics and market share of Unix for websites, 2016, https://w3techs.com/technologies/details/os-unix/all/all

[11]: Sharath Venkatesha, Shatrugna Sadhu, Sridhar Kintali, Survey of Virtual Machine Migration Techniques, 2011, https://www.academia.edu/760613/Survey_of_Virtual_Machine_Migration_Techniques

[12]: Dustin Sklavos, DDR3 vs. DDR4: Four Generations, Raw Bandwidth by the Numbers, 2015, http://www.corsair.com/en-us/blog/2015/september/ddr3_vs_ddr4_generational

[13]: , Protecting Mission-Critical Workloads with VMware Fault Tolerance, 2009,

https://www.vmware.com/files/pdf/resources/ft_virtualization_wp.pdf
[14]: Isaku Yamahata, Qemu-devel] [PATCH v4 2/2] umem: chardevice for kvm postcopy, 2012, https://lists.gnu.org/archive/html/qemu-devel/2012-10/msg05274.html
[15]: Andrea Arcangeli, madvise(MADV_USERFAULT) & sys_remap_anon_pages() , , https://lwn.net/Articles/549503/
[16]: Eric Harney, Sebastien Goasguen, Jim Martin, Mike Murphy, Mike Westall , The Efficacy of Live Virtual Machine Migrations Over the Internet , 2007, https://pdfs.semanticscholar.org/127e/1a2b3db1dd85e3244a2450bb93a8b94154b3.pdf
[17]: Fabian Beck, Stephan Diehl, On the congruence of modularity and code coupling, 2011
[18]: Barry Boehm, SPECIAL REPORTCMU/SEI-2000-SR-008Spiral Development:Experience, Principles,and Refinements, 2000, https://www.sei.cmu.edu/reports/00sr008.pdf
[19]: Linus Torvalds, Re: clone() <-> getpid() bug in 2.6? , 2004, https://lkml.org/lkml/2004/6/5/88
[20]: CRIU, 2017, https://criu.org/
[21]: Andrew Morton, Merge branch 'akpm' (aka "Andrew's patch-bomb, take two"), 2012, https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=099469502f62fbe0d7e4f0b83a2f22538367f734

A. Appendix

A.1 Command-line Interaction Examples

A.1.1 Show the command-line help and debugging output

Start our target process, countforever which increments and prints forever:

```
user@system:~/testdir$ ./countforever
0
1
2
3
4
5
6
7
8
...
```

In another terminal session: Show command-line help:

```
user@system:~/testdir$ ./pfrez --help
Usage of ./pfrez:
  -compress string
      Compression mode: none | gzip | flate | snappy (default "none")
  -debug
      Debug: true | false, if enabled outgoing data will be displayed
  -dest string
      Output sink: stdout | tcp|udp:host:port | unix:socketpath |
snapshot-filepath (default "stdout")
  -dial-timeout duration
      Optional: Duration to wait for socket level connection to be
established
  -encrypt string
      Encryption mode: none | AES-CFB|AES-CTR|AES-OFB:keypath (default
"none")
  -halt
      Halt the target process after state capture and transmission is
complete
  -pid int
      PID of process to be frozen (default -1)
  -write-timeout duration
      Optional: Duration to wait transmitting data to an active stream
before timing out
```

Demonstrate pfrez must run as root!

```
user@system:~/testdir$ ./pfrez
pfrez: This utility must be executed as root.
```

Show state capture with debug as the output sink:

```
user@system:~/testdir$ sudo ./pfrez -pid=`pgrep countforever` -debug
Process Name: ./countforever
Process Identifer (PID): 4173

Registers:
0 Cs: 0x33  1 Ds: 0x0   2 Eflags: 0x246   3 Es: 0x0
4 Fs: 0x63  5 Fs_base: 0x1FB9880    6 Gs: 0x0   7 Gs_base: 0x0
8 Orig_rax: 0x1   9 R10: 0x22 10 R11: 0x246     11 R12: 0x6C0300
12 R13: 0x8 13 R14: 0x1 14 R15: 0x7FFC406E9018  15 R8: 0x4A67C0
16 R9: 0x0  17 Rax: 0x7 18 Rbp: 0x7FCFE50A8000  19 Rbx: 0x8
20 Rcx: 0xFFFFFFFFFFFFFFFF    21 Rdi: 0x1 22 Rdx: 0x8 23 Rip: 0x433F70
24 Rsi: 0x7FCFE50A8000  25 Rsp: 0x7FFC406E8978  26 Ss: 0x2B

Memory Map Metadata & Content MD5 hashes:
 0 Meta: 400000-4bf000  r-xp  0     8:1   9971119     /home/undriedsea/MSE
Captstone/Source/mse/src/testprogs/countforever
 0 Hash: B4FA1E03B5A2620FD3086A177063F7E8
 1 Meta: 6bf000-6c2000  rw-p  bf000 8:1   9971119     /home/undriedsea/MSE
Captstone/Source/mse/src/testprogs/countforever
 1 Hash: 2445E726046AD4998E9D0B2E23C34553
 2 Meta: 6c2000-6c5000  rw-p  0     0:0   0
 2 Hash: 1277FEAFC34FD7D9FEF1F7F1886CE51F
 3 Meta: 1fb9000-1fdc000      rw-p  0     0:0   0     [heap]
 3 Hash: B1C74438B7E237BD47657D13D3A88AA7
 4 Meta: 7fcfe50a8000-7fcfe50a9000  rw-p  0     0:0   0
 4 Hash: DE814B82B447A17E004A7AEF4D29089A
 5 Meta: 7ffc406ca000-7ffc406eb000  rw-p  0     0:0   0     [stack]
 5 Hash: 4D780E9590E9B320E0A96F7B97B45C2F
 6 Meta: 7ffc407f1000-7ffc407f3000  r-xp  0     0:0   0     [vdso]
 6 Hash: 8676565460AC246B4CCB11546EEE4322
 7 Meta: ffffffffff600000-ffffffffff601000      r-xp  0     0:0   0
      [vsyscall]
 7 Hash: 6A5113432CAED2E2B046E7C7B6D2A0F8

Open File Handles:
File handle: 0, Path: "/dev/pts/0", Type/Mode: 410000620 (pseudo-terminal:
Dcrw--w----), Seek position: 0, Flags: 32770
File handle: 1, Path: "/dev/pts/0", Type/Mode: 410000620 (pseudo-terminal:
Dcrw--w----), Seek position: 0, Flags: 32770
File handle: 2, Path: "/dev/pts/0", Type/Mode: 410000620 (pseudo-terminal:
Dcrw--w----), Seek position: 0, Flags: 32770
```

A.1.2 Create a snapshot and restore from it

This is a simple example using stdin/stdout for input/output sinks.

Note: Our ./countforever process is still running from A.2.1.

Capture process state and send it to stdout :

```
user@system:~/testdir$ sudo ./pfrez -pid=`pgrep countforever` > demo.snap
```

Now, restore process from above snapshot (via stdin) :

```
user@system:~/testdir$ ./pthaw < demo.snap
.Attaching... .Attached.
Loading Registers... Loaded.
Resuming process...
32218008
32218009
32218010
32218011
32218012
32218013
...
```

A.1.3 Show the use of compression and source destination specifiers

This is a simple example that address compression while using explicit source destination specifiers (rather than stdin/stdout) for input/output sinks.

Note: Our ./countforever process is still running from A.2.1.

Capture process state and send to gizipped file :

```
user@system:~/testdir$ sudo ./pfrez -pid=`pgrep countforever`
-compress=gzip -dest=demo.snap.gz
```

Restore process from gzipped snapshot file :

```
user@system:~/testdir$ ./pthaw -src=demo.snap.gz
.Attaching... ..Attached.
Loading Registers... Loaded.
Resuming process...
4230339
4230340
4230341
4230342
4230343
4230344
4230345
...
```

A.1.4 Perform local capture and restore (without an intermediate snapshot)

Capture the process and restore the to same host; a clone operation.

Note: Our ./countforever process is still running from A.2.1.

Capture process state and send via pipe to pthaw to be restored :

```
user@system:~/testdir$ sudo ./pfrez -pid=`pgrep countforever` | ./pthaw
.Attaching... .Attached.
Loading Regsisters... Loaded.
Resuming process...
2006890
2006891
2006892
2006893
2006894
2006895
...
```

A.1.5 Demonstrate getpid system-call interception

Start our test program that makes getpid/getppid system-calls; getpid is intercepted but getppid is not. This program will print if either change.

```
user@system:~/testdir$ sudo ./getpid
This process's PID is 4393.
This process's parent's PID is 4090.

No changes in PID or PPID in last 10 seconds.
No changes in PID or PPID in last 10 seconds.
No changes in PID or PPID in last 10 seconds.
```

Capture and restore locally:

```
user@system:~/testdir$ sudo ./pfrez -pid=`pgrep getpid` | ./pthaw
.Attaching... Attached.
Loading Regsisters... Loaded.
Resuming process...
This process's parent's PID changed from 4090 to 4395!

No changes in PID or PPID in last 10 seconds.
```

The printed change in getppid demonstrates that cloning the process resulted in an environmental change, but lack of a printed getpid change shows that system-call interdiction is functioning for that system call.

A.1.6 Demonstrate getpid and read/write system-call interception

Start our program that makes read/write system calls by opening the same file twice, one for reading and once for writing in append mode. It print the contents of the file, then appends to it:

```
user@system:~/testdir$ ./files
Reading and appending to: files_output.txt

files_output.txt contains:
Hello world #0!

files_output.txt contains:
Hello world #0!
Hello world #1!
Hello world #1!
```

Show file handles using the -debug flag:

```
user@system:~/testdir$ sudo ./pfrez -pid=`pgrep files`-debug | grep 'File
handle'
File handle: 0, Path: "/dev/pts/0", Type/Mode: 410000620 (pseudo-terminal:
Dcrw--w----), Seek position: 0, Flags: 32770
File handle: 1, Path: "/dev/pts/0", Type/Mode: 410000620 (pseudo-terminal:
Dcrw--w----), Seek position: 0, Flags: 32770
File handle: 2, Path: "/dev/pts/0", Type/Mode: 410000620 (pseudo-terminal:
Dcrw--w----), Seek position: 0, Flags: 32770
File handle: 3, Path: "/home/testenv/files_output.txt", Type/Mode: 664
(regular file: -rw-rw-r--), Seek position: 80, Flags: 33793
File handle: 4, Path: "/home/testenv/files_output.txt", Type/Mode: 664
(regular file: -rw-rw-r--), Seek position: 80, Flags: 32768
```

We can see the file handles to /home/testenv/files_output.txt for reading and writing are file descriptors 3 and 4.

Capture and restore locally:

```
user@system:~/testdir$ sudo ./pfrez -pid=`pgrep files` | ./pthaw
.Attaching... .Attached.
Loading Regsisters... Loaded.
Resuming process...

files_output.txt contains:
Hello world #0!
Hello world #1!
Hello world #1!
Hello world #2!

files_output.txt contains:
Hello world #0!
Hello world #1!
Hello world #1!
Hello world #2!
Hello world #2!
Hello world #3!
```

Observe that the new files handles (5 & 6) do not match the original ones (3 & 4):

```
user@system:~/testdir$ ls -lah /proc/`pgrep ploader`/fd/
total 0
dr-x------ 2 undriedsea undriedsea  0 Jan 11 18:28 .
dr-xr-xr-x 9 undriedsea undriedsea  0 Jan 11 18:27 ..
lr-x------ 1 undriedsea undriedsea 64 Jan 11 18:28 0 -> pipe:[34116]
lrwx------ 1 undriedsea undriedsea 64 Jan 11 18:28 1 -> /dev/pts/14
lrwx------ 1 undriedsea undriedsea 64 Jan 11 18:28 2 -> /dev/pts/14
lr-x------ 1 undriedsea undriedsea 64 Jan 11 18:28 3 -> pipe:[34118]
l-wx------ 1 undriedsea undriedsea 64 Jan 11 18:28 4 -> pipe:[34119]
l-wx------ 1 undriedsea undriedsea 64 Jan 11 18:28 5 ->
/home/undriedsea/MSE Captstone/Source/mse/src/testenv/files_output.txt
lr-x------ 1 undriedsea undriedsea 64 Jan 11 18:28 6 ->
/home/undriedsea/MSE Captstone/Source/mse/src/testenv/files_output.txt
```

However, since the cloned program continues to function, we know read/write system call interdiction must be taking place. *Note: You can see the file handles above (3 & 4) which are used by pthaw/ploader communication during restoration.*

A.1.6 Migration: Local Capture and Remote Restore

Now we will capture a process and migrate to a new host with fast compression and encryption.

First we must generate our AES key and since AES is a form a symetric encryption we need to distribute this preshared key to both systems.

```
user@system:~/testdir$ dd if=/dev/urandom of=./preshared.key ibs=1
count=32
#SCP to other host...
```

Start our target process, countforever which increments and prints forever:

```
user@system:~/testdir$ ./countforever
0
1
3
```

Start listening state-sink/supervisor on remote host:

```
./pthaw -src=tcp:25088
```

Invoke capture command (pfrez) and instruct it transmit state with fast compression, AES encryption using our pre-shared key and to halt the target process after capture:

```
sudo ./pfrez -pid=`pgrep countforever` -compress=snappy -encrypt=AES-
CTR:preshared.key -dest=tcp:192.168.0.2:25088 -halt
```

Observe the original process is no longer running:

```
...
338155
338156
338157
338158
338159
338160
Killed
```

Observe the remote hosts' pthaw has started the clone:

```
.Attaching... .Attached.
Loading Regsisters... Loaded.
Resuming process...
338161
338162
338163
338164
338165
338166
...
```