Enhancing Cloud Security and Privacy: The Unikernel Solution

Alfred Bratterud
Dept. of Computer Science
Oslo and Akershus University
Oslo, Norway
Email: alfred.bratterud@hioa.no

Andreas Happe
Dept. Digital Safety & Security
Austrian Inst. of Tech. GmbH
Vienna, Austria
Email: andreas.happe@ait.ac.at

Bob Duncan
Computing Science
University of Aberdeen
Aberdeen, UK
Email: bobduncan@abdn.ac.uk

***Abstract*—Cloud security and privacy is a very challenging problem to solve. We started a project to explore a new approach to addressing this problem by utilising a unikernel based solution. In this paper, we outline the technical details of such an approach, identifying how this new approach can better address the issues involved. We have demonstrated how this new approach can improve the status quo.**

***Index Terms*—Cloud security and privacy; management control; compliance; complexity**

I. Introduction

In [1], we provided a high level account of ten security issues, which management (Mgt) need to take account of when using cloud computing systems, and suggested how unikernel-based systems might address many of those issues, as we see in TABLE I below.

TABLE I. ITEMS ADDRESSED BY UNIKERNELS ©2016 [1]

Issue	Description	Helped by:
1	Definition of security goals	Mgt and Unikernels
2	Compliance with standards	Mgt and Unikernels
3	Audit issues	Mgt and Unikernels
4	Management approach	Mgt and Unikernels
5	Technical complexity of cloud	Unikernels
6	Lack of responsibility and accountability	Mgt/Cloud Service Providers
7	Measurement and monitoring	Unikernels
8	Management attitude to security	Mgt
9	Security culture in the company	Mgt
10	Threat environment	Unikernels can help to enforce good design/architectural decisions

While these security issues can be successfully addressed by other means, the reality, as evidenced by the recurring success of attackers, is that many companies are failing to apply the necessary rigour needed to resolve these issues in their existing approaches. Year after year, many attacks, which are both simple and relatively inexpensive to defend against, continue to be exploited.

In this paper, we introduce a framework of definitions and metrics for classifying unikernel systems, which we later make use of in the design, testing and assessment of new unikernel based system architectures. We compare a number of other unikernel and microkernel systems in this paper, but because of space constraints, do not address every single system available. In Section II, we discuss the background, motivation for this work, work already carried out and future work proposed. In Section III, we outline some necessary definitions and preliminary observations on unikernels, and in Section IV we consider 6 security observations relating to unikernels. The remainder of the paper is organized as follows: in Section V, we review the relationship between unikernels and microkernels; in Section VI, we discuss the implications of implementation language choice; in Section VII, we present well defined properties of unikernel systems; in Section VIII, we outline our proposed solution. In Section IX, we show how our proposed approach will address those key issues identified in TABLE: I. In Section X, we consider some initial thoughts on attack vectors; and in Section XI, we discuss our conclusions.

II. Background, Motivation, Work Already Carried out and Proposed Future Work

The authors share a common interest in finding a solution to the challenging problem of cloud cyber security. The minimal resource efficiency of IncludeOS motivated the authors to consider whether there might be a possibility to develop a framework, based on unikernels, to deliver a far more secure system that could be run on cloud, and would offer high levels of security, privacy, audit and forensic trails, good scalability, high resource efficiency, and have the ability to take away the prospect of mis-configuration by users through lack of understanding of how to configure much more complex systems.

In [1], we outlined how the concept might be developed, and in [2], we considered how the proposed framework might be adapted to incorporate the Internet of Things (IoT). This paper outlines in a more formal way, definitions and metrics for classifying unikernel systems, which will be referred to in future work involving design, testing and assessment of new unikernel based system architectures. We next extend the work carried out in [1][2], providing much more detail on how the IoT system might be developed, and further examine how the single responsibility inherent in the unikernel design could be harnessed to provide a far more robust defence against common security problems.

 ISBN: 978-1-61208-529-6

As we develop each part of the framework, we carry out in-house penetration testing to ensure the robustness of the approach. Each part developed is intended to work seamlessly with all of the previous parts, so that the system as a whole will work properly as it is developed and grown. Once it reaches the point where it will interest at enterprise level, we will carry out large scale empirical testing to assess what will happen in the real world. We are developing fuzzing based penetration approaches, adapting tools and sanitizers, hardening tools and whatever else we can use to strengthen the user environment. We have still to develop communication channels, proper and secure audit and forensic trails, specialised storage, and plan to ensure the framework is capable of working under both object oriented styles and the model view controller paradigm.

III. A PRECURSOR TO FORMAL WORK ON UNIKERNELS

Modern unikernel research, particularly concerning deployment in cloud, is still in its infancy and the literature currently available does not include much in the way of theoretical work or precise definitions, but rather takes a pragmatic approach [3]–[5]. One popular definition states that *"Unikernels are specialised, single-address-space machine images constructed by using library operating systems*; and that *Unikernels provide many benefits compared to a traditional Operating System (OS), including improved security, smaller footprints, more optimisation and faster boot times"* [6]. Without further qualification, directly associating unikernels with security benefits can be hazardous. It is not at all clear what "machine images" or "operating system libraries" are, nor what it means to construct them, and hence it is unclear precisely how unikernels can be said to be more secure. It is our view that stricter definitions are both necessary and achievable. We propose a set of working definitions intended to make the following exposition more precise, and to serve as a theoretical basis of a framework for unikernel based cloud computing.

To this end we can go back over four decades to early work on virtualization of mainframes. Early single address space operating systems, such as Mungi or Opal and many others do not seem to have much traction today. In 1995 [7], there was some early work on the Exokernel system, with some updating over the years, but little widespread use. Microsoft work on library operating systems, Drawbridge [8], saw little use at the time other than for research. It later evolved into the Haven system [9], which was intended for use in the Azure cloud, and also was integrated into Windows 10 as part of the pico-process security architecture.

Definition III.1. Popek-Goldberg virtual machine. An environment created by the virtual machine monitor, which is functionally equivalent to the physical machine on a given hardware (HW) platform, as defined in [10].

Popek and Goldberg provide formal definitions of both Virtual Machine Monitor and Virtual Machine, which form the most well known, if not the only, formally defined virtualization platform, which also directly corresponds to modern HW and nomenclature. An x86 Popek-Goldberg virtual machine is a virtual machine running under x86 Popek-Goldberg compliant HW virtualization, e.g., x86 HW with *vt-x* extensions. To give a precise definition of unikernels, we need to define their constituent parts. Our intention is not to provide definitions that encompass all the complexities or functionalities of unikernels, but rather the opposite; just enough to get a precise idea of what unikernels are and what they are not.

Definition III.2. Compiled program object, symbol. An n-tuple of machine instructions from a Turing complete instruction set, e.g., Intel x86, or an arbitrary sequence of bytes b, i.e., $0 >= b < 2^8$.

Definition III.3. Software library, symbol. A *Software library* is taken to be a collection of compiled program objects, $\{O_1, ..., O_m\}$ and arbitrary byte-sequences (e.g., data) $\{D_{m+1}, ..., D_n\}$ providing symbol resolutions, i.e., linkable objects, each corresponding to a *symbol* in the set $\{S_1, ..., S_m, S_{m+1}, ..., S_n\}$

The intuition here is to capture the idea of a library of compiled code, where functions and data are represented as binary objects in, e.g., `libos.so`, `libos.a`, `libos.dll` etc., where functions and static data can be accessed via symbols available to a linker. For the purposes of this paper, we assume the process of linking and symbol resolution are given and well defined.

Definition III.4. Software service. Objects, and optionally symbols, from a software library, with the addition of an entry point object O_0 corresponding to a symbol S_0 (e.g.,`_start`) that may or may not be present in the service, providing the compiled code necessary for a program to be executable on a given HW architecture.

Compiling an executable binary typically includes defining the entry point, e.g., `main` in ISO C, from where to start executing functions provided from a library. Compilers such as `gcc` will typically pre-pend some additional functionality through, e.g., the `_start`-symbol, mapped to code for initializing the C-runtime, including calling global constructors, zero-initializing the `.bss`-segment etc. The symbols are required during link-time but can later be stripped out when they are mapped to memory addresses relative to the binary.

We can now give an operational definition of a library operating system. The term has held different meanings [11][3], and we do not intend the following to be canonical, but merely one that will suffice to precisely define a unikernel for the purposes of our framework.

Definition III.5. Library operating system. For a software service SW, where the objects $\{O_0, ..., O_n\}$ form the set of objects necessary and sufficient for SW to run on a given HW platform, a library operating system is a software library that can provide $\{O_0, ..., O_n\}$

This definition implies that a library operating system provides all the objects necessary to form a fully functional program, independent of any other software present on a sys-

 ISBN: 978-1-61208-529-6

tem, except that which may or may not be presented through an instruction level interface, e.g., software that responds to a *trap* on the *Virtual Machine Monitor* in the Popek-Goldberg model.

1) Definition of a unikernel: In the context of virtual machines and cloud computing, it makes sense to describe the whole virtual machine as a unikernel [3], as there is in fact no classic boundary between kernel- and user space, and also because any combination of objects that can be pulled from the library operating system individually can be combined with a piece of software to form a unique whole. This piece of completely linked software will have full access to HW on the same level as a classic kernel.

In the context of classic operating system kernels, however, the library operating system designed to produce unikernels may also be called a unikernel [12] in reference to "micro-kernel", "nanokernel", "monolithic kernel" etc. It could be argued that if all the contents of a virtual machine were to be considered a unikernel, there wouldn't really be any point in using the word "kernel".

The following definiton is intended to be sufficiently flexible to allow both interpretations.

Definition III.6. Unikernel. Given a library operating system OS, a unikernel U is defined as $U \subseteq OS$ such that U is sufficient and necessary to provide complete linkage to some service S for a given HW platform.

Using an inclusive subset allows both the whole library operating system and any subset to be called a unikernel.

Definition III.7. Unikernel machine image A software service $SW, \cup U$ where U is the unikernel for SW, they both share the same address space, and with the addition of any facilities necessary to start SW on a given well-defined virtualization platform, e.g., a bootloader in the case of an x86 Popek-Goldberg virtual machine.

Definition III.8. Popek-Goldberg unikernel. A Popek-Goldberg virtual machine initialized with a unikernel machine image.

IV. Six security observations in Unikernel-based systems

We have identified 6 security observations, which are exhibited by unikernel systems:

- Choice of service isolation mechanism;
- The concept of reduced software attack surface;
- The use of a single address space, shared between service and kernel;
- No shell by default, and the impact on debugging and forensics;
- Micro services architecture and immutable infrastructure;
- Single thread by default.

A. Choice of service isolation mechanism

In the previous paper in this series, the argument was made for why classic virtualization is the preferred platform for secure cloud computing. While many alternatives exist, which are both practical and widely trusted, one cannot reason precisely about their security properties unless they are well defined. In this paper, we make no judgements about their usefulness, but merely note that classic virtualization has had a precise foundation since 1974. We believe that the lack of similar models for other modes of virtualization is due to the fact that Popek-Goldberg virtualization exists at the instruction level, which is necessarily simple in nature as it must be implemented in physical circuitry. Other approaches typically rely on higher level software interfaces, and are thus harder to define precisely. Despite the simplicity of C, it still proves a hard nut to crack for the purposes of formal verification [13].

B. Reduced software attack surface

Using the above definitions we can now define the software attack surface of a system as the sum of all objects, in bytes.

Definition IV.1. Software Attack surface. The number of bytes in a system, physically available for reading, writing or executing as instructions for a given HW architecture.

Physical protection can be seen as a grey area when it comes to microcode, firmware and otherwise mutable HW such as, e.g., field-programmable gate arrays (FPGAs). This definition is intentionally kept general in order to allow further specifications to refine the meaning of "physically available" for a given context. The following example can serve to illustrate how the definition can be used for one of many purposes. Building a classic virtual machine (VM) using Linux implies simply installing Linux, and then installing the software on top. Any reduction in software attack surface must be done by removing unneeded software and kernel modules (e.g drivers). Take TinyCore Linux as an example of a minimal Linux distribution and assume that it can produce a machine image of 24 MegaBytes (MB) in size.

Given this intuition, let L be a collection of compiled program objects for x86, such that $L = \{O_1, ..., O_{2400}\}$, i.e., all the objects provided by TinyCore Linux, totalling 24MB in size, and for simplicity assume the objects are uniformly sized, 1Kb each. Adding a 1MB software service SW, which require $\{O_0, ..., O_{1000}\}$ to be executable, we get a software attack surface of 25MB, regardless of how many objects of L were actually needed by SW. Assuming a *library operating system* existed that could provide $\{O_0, ..., O_{1000}\}$, a *unikernel* would by definition III.6, provide exactly those objects, forming a 2MB sized unikernel machine image. (2MB + 512 bytes of bootloader code in an x86 HW-VM). Hence the software attack surface of the unikernel VM is reduced by 92%. Conversely, the Linux VM could be said to add 23/1 * 100 = 2300% of unnecessary code, which we will refer to as *bloat* or *increased software attack surface* depending on context.

C. The use of a single address space

The main objective for postulating a single address space is to imply single process or singular purpose. In a classic kernel, the need for multiple address spaces is prompted by the need to run multiple processes, which must be kept separate to ensure consistency and integrity among them. A classic

kernel will typically rely on virtual memory implemented in HW to ensure process isolation and to provide a controlled virtual to physical address translation. Popek-Goldberg virtualization relies directly on this general concept without further extension. Aside from the performance degradation often seen in nested address translation, we take the view that introducing virtual memory and multiple processes inside a Popek-Goldberg virtual machine needlessly complicates an already complex system. In particular, it lays upon the virtual machine the added responsibility of creating and maintaining a process-kernel boundary.

Definition IV.2. Single address space. For a computer system, a *single address space* is defined as an interval of positive integers $[a_0, .., a_n]$ where n is a power of two, representing the total addressable memory of a system in a given state, such that dereferencing any address $*a_x$ from anywhere inside the system would access the same physical memory cell.

The intuition is that virtual memory is not employed inside the system, effectively eliminating the possibility of running several disjoint processes. We are not making any assumptions or requirements as to whether or not all addresses are in fact accessible, e.g., physically present or readable / writeable / executable, merely that they point to the same location if any. Note that virtual memory can and will be employed on the virtual machine monitor, to protect one VM from another, but further nesting of virtual memory would violate the single address space principle.

D. No shell by default and the impact on debugging and forensics

One feature of unikernels that immediately makes it seem very different from classic operating systems is the lack of a command line interface. This is however a direct consequence of the fact that classic POSIX-like command line interpreters (CLI)s are run as a separate process (e.g., `bash`) with the main purpose of starting other processes. Critics might argue that this makes unikernels harder to manage and "debug", as one cannot "log in and see what's happened" after an incident, as is the norm for system administrators. We take the position that this line of argument is vacuous; running a unikernel rather corresponds to running a single process with better isolation, and in principle there is no more need to log in to a unikernel than there is to log in to, e.g., a web server process running in a classic operating system.

While unikernels by definition are a single address space virtual machine, with no concept of classic processes, a text based CLI could be provided (e.g., IncludeOS does provide an example) — the commands wouldn't start processes, but call functions inside the program. From a security perspective we take the view that this kind of ad-hoc access to program objects should be avoided. While symbols are very useful for providing a stack trace after a crash or for performance profiling, stripping out symbols pointing to program objects inside a unikernel would make it much harder for an attacker to find and execute functions for malicious and unintended purposes. Our recommendation is that this should be the default mode for unikernels in production mode. We take the view that logging is of critical importance for all systems, in order to provide a proper audit trail. Unikernels however simply need to provide the logs through other means, such as over a virtual serial port, or ideally over a secure networking connection to a trusted audit trail store.

Lastly it is worth mentioning that unikernels in principle have full control over a contiguous range of memory. Combined with the fact that a crashed VM by default will "stay alive" as a process from the virtual machine manager (VMM) perspective, and not be terminated, this means that in principle the memory contents of a unikernel could be accessed and inspected from the VMM after the fact, if desired. Placing the audit trail logs in a contiguous range of memory could then make it possible to extract those logs also after a failure in the network connection or other I/O device normally used for transmitting the data. Note that this kind of inspection requires complete trust between the owner of the VM and the VMM (e.g., the cloud tenant and cloud provider). Our recommendation would be not to rely on this kind of functionality in public clouds, unless all sensitive data inside the VM is encrypted and can be extracted and sent to the tenant without decrypting it.

E. Micro services architecture and immutable infrastructure.

Micro services is a relatively new term founded on the idea of separating a system into several individual and fully disjoint services, rather than continuously adding features and capabilities to an ever growing monolithic program. Being single threaded by default unikernels naturally imply this kind of architecture; any need for scaling up beyond the capabilities of a single CPU should be done by spawning new instances. While classic VM's require a lot of resources and impose a lot of overhead, minimal virtual machines are very lightweight. As demonstrated in [14] more than 100,000 instances could be booted on a single physical server and [12] showed that each virtual machine, including the surrounding process require much less memory than a single "Hello World" Java program running directly on the host.

An important feature of unikernels in the context of micro services is that each unikernel VM is fully self contained. This also make them immune to breaches in other parts of the service composition, increasing the resilience of the system as a whole. Add to this the idea of *optimal mutability* (defined below), and each unikernel-based micro service can in turn be as immutable as is physically possible on a given platform. In the next paper in this series we expand upon these ideas and take the position that composing a service out of several micro services, each as immutable as possible, enables overall system architects and decision makers to focus on a high level view of service composition, not having to worry too much about the security of their constituent parts. We take the view that this kind of separation of concerns is necessary in order to achieve scalable yet secure cloud services.

 ISBN: 978-1-61208-529-6

F. Single threaded by default

While the above definitions do not impose any restrictions on whether or not a unikernel can run several concurrent threads or multiple CPU cores, it is well known that concurrency is a major source of errors accounting for a significant number of vulnerabilities. IncludeOS and MirageOS are both examples of unikernels that are single threaded by default. Efficiency is achieved by event based asynchronous interfaces with no blocking calls. While pre-emptive interrupt handling and concurrency using shared memory are necessary for certain workloads, we take the view that single threaded concurrency free services are by nature less complex and thus less error prone. It is also well known that threaded applications perform worse inside virtual machines than single threaded applications due to the extra layer of context switches necessary to schedule threads inside the VM as well as outside.

Our recommendation is to keep unikernels single threaded by default and rather achieve concurrency by adding more instances, to the extent possible. In a modular library OS one can add threading and re-entrant versions of libraries as optional components without causing bloat or increased complexity to unikernels not requiring concurrency.

V. Relationship to microkernels

While there exists a rich fauna of operating system kernel types, the most well known distinction is between monolithic kernels and micro kernels. For this reason we'll briefly explain how unikernels fit in this spectrum. Microkernel operating systems are absolutely minimal in the sense that nothing that doesn't have to be in the kernel is. However, most implementations such as the L4 are still A) multi-process; B) not library operating systems; and C) will typically have a classic style command line, etc., which would make it almost orthogonal to our purpose as it addresses other issues (L4 is focussed mainly on fast Industrial PCs). That being said, they have an advantage over classic kernels when it comes to: A) software attack surface (it can run many programs, but it does not have to); and B) complexity. The simplicity of the microkernel is what made it possible to do formal verification of the Haskell implementation of L4 - and that is a major security benefit.

Our position is that unikernels have the potential to incorporate the "small and simple" from microkernels, while still adding new security features — in particular: 1) The library operating system approach, which guarantees a minimal amount of unnecessary code is introduced; 2) the single-purpose approach; and 3) it is single-threaded by default. This provides a further means of simplification (parallel programming is notoriously error-prone), while also strongly encouraging micro service architecture, which increase resilience of the system as a whole.

VI. Choice of implementation language

Definition VI.1. Independent systems language. A Turing complete programming language with facilities to utilize the whole instruction set for a given HW architecture, including writing arbitrary data to arbitrary addresses.

C and C++ are examples of independent systems languages for most modern HW architectures, e.g., x86: the `asm` keyword (i.e., "inline assembly") makes the full instruction set available to the programmer, including privileged instructions such as `hlt`,`in` and `out`, and the pointer data type and (unsafe) type conversion allows arbitrary data to be written to arbitrary addresses. Type safe languages such as javascript, OCaml, Haskell and Python are not independent systems languages by design; type safety can be immediately violated by, e.g., type coercion. The requirement for being an independent systems language is thus incompatible with type safety. To bridge this incompatibility, unikernels written in type safe languages must necessarily contain a portion of code written in an independent systems language. In most cases, such as with MirageOS, this is done in C.

VII. Well defined properties of unikernel systems

Based on the previous definitions we can now provide a framework of well-defined properties of unikernel-based systems:

- **Service isolation:**. A well-defined and absolute isolation mechinanism such as Popek-Goldberg virtualization is preferable as 0 bytes of code needs to be shared between services during runtime. Enforcement is performed by HW at the instruction level. Following closely is Xen PVH, which is mostly HW virtualization, but some shared code. Paravirtualization shares a thin yet fairly complex set of software bindings and HW is only used for classic process isolation. One way to quantify this property is the amount of software, in bytes, shared by each service on the virtual machine monitor. Microcode / firmware would be a grey area, but that would be common to all current isolation mechanisms.
- **VM Slimness, Bloat and software attack surface:** For a software service SW, requiring objects A, B and C to form a machine image, a system (e.g., unikernel library) that can produce a virtual machine containing exactly $\{SW, A, B, C\}$ without $\{D, E, F, ...\}$ provides *optimal slimness*. Conversely, the amount of code added to the VM in addition to $\{SW, A, B, C\}$ adds *bloat*. As an example, if $\{SW, A, B, C\}$ was 10MB in size and an operating system added 5 MB that would be 50% of bloat. Linux-based virtual machines would easily be 2000% bloated for single-purpose virtual machines, e.g., using a trimmed down Linux micro-core of 24MB used to run a 1MB service, 96% of the machine image would be operating system. IncludeOS instances are typically 1 MB of OS, so if the service could run with IncludeOS that would be 50/50 software and OS, plus a few percent overweight (one typically would not use all the code included, even if one included only the objects needed, unless the objects themselves are each absolutely minimal). Given that 1MB was sufficient to add the required operating system parts, wrapping it in a Linux VM would literally make for 2300% of bloat.

 ISBN: 978-1-61208-529-6

- **System mutability:** To what extent is it possible to change the system once launched? This is hard to quantify, but we propose the following set of properties as "optimal immutability", which a system should strive for:
  1) All data that can be read-only is;
  2) All executable code is write-protected;
  3) Write-able areas of memory (i.e., the heap / working memory) is not executable.

  Enforcement of these rules should be implemented at the lowest level. In IncludeOS, this kind of protection cannot really be enforced on current platforms. Type-safe language unikernels, such as Mirage, have a certain degree of language-level protection, but only in the parts of the unikernel not written in C. We propose a future work on hypervisors where we provide an interface for specifying which parts of the VM that should be read-only, execute- and read/write (but not execute), when the system boots. This way, the hypervisor at ring -1 can set up memory segments inside the VM before it starts, denying even the VM itself the ability to modify read-only parts of memory. Having the CPU enforce these rules will make it useless to inject code into a VM, if one found a way to do it, as jumping to that code would trigger a HW trap.
- **Possibility of internal system misuse:** To what extent does the operating system allow parts of the code to be used for unintended purposes? Having a terminal makes several commands available for "general purpose" or "ad-hoc use" of the code embedded into the system. Not having a terminal, or other similar means of allowing ad-hoc function calls, greatly reduces or entirely removes this possibility.

VIII. Our proposed solution

By default, in the interests of usability, conventional systems open many more ports than may be needed to run a system. An open port, especially one that is not needed, is another route in for the attacker. We also take the position that the probability of vulnerabilities being present in a system increases proportionally to the amount of executable code it contains. Having less executable code inside a given system will reduce the chances of a breach and also reduce the number of tools available for an attacker once inside. As Meireles [15] said in 2007 *"... while you can sometimes attack what you can't see, you can't attack what is not there!"*. Given the success with which the threat environment continually attacks business globally [16]–[20], it is clear that many companies are falling down on many of the key issues we have highlighted in Section I. It is also clear that a sophisticated and complex solution is unlikely to work. Thus we must approach the problem from a more simple perspective.

A. Service isolation

A fundamental premise for cloud computing is the ability to share HW. In private cloud systems, HW resources are shared across a potentially large organization, while on public clouds, HW is shared globally across multiple tenants. In both cases, isolating one service from the other is an absolute requirement.

The simplest mechanism for service isolation is simply *process isolation* in classic kernels, relying on HW supported virtual memory, e.g., provided by the now pervasive x86 protected mode. While process isolation has been used successfully in mainframe setups for decades, access to terminals with limited user privileges has also been the context for classic attack vectors such as stack smashing, root-kits etc., the main problem being that a single kernel is being shared between several processes and that gaining root access from one terminal would give access to everything inside the system. As a result, much work was done in the sixties and seventies to find ways to completely isolate a service without sharing a kernel. This work culminated with the seminal 1974 paper by Popek and Goldberg [21] where they present a formal model describing the requirements for complete instruction level virtualization, i.e., *HW virtualization*.

While HW virtualization was in wide use on e.g., IBM mainframes from that time, it wasn't until 2005 that the leading commodity CPU manufacturers, Intel and AMD, introduced these facilities into their chips. In the meantime, paravirtualization had been re-introduced as a workaround to get virtual machines on these architectures, notably in [22]. While widely deployed and depended upon, the Xen project has recently been evolving its paravirtualization interface towards using HW virtualization in, e.g., PVH [23] stating that *"PVH means less code and fewer Interfaces in Linux/FreeBSD: consequently it has a smaller TCB and software attack surface, and thus fewer possible exploits"* [24].

Another isolation mechanism is operating system-level virtualization with containers, e.g., Linux Containter (LXC) popularized in recent years by Docker, where each container represents a userspace operating environment for services that all share a kernel. The mechanism for isolating one container from another is classic process isolation, augmented with software controls such as cgroups and Linux namespaces. While containers do offer less overhead than classic virtual machines, a good example where containers make a lot of sense would be trusted in-house clouds, i.e., Google is using containers internally for most purposes [25]. We take the position that HW virtualization is the simplest and most complete mechanism for service isolation, with the best understood foundations as formally described by Popek and Goldberg, and that this should be the preferred isolation mechanism for secure cloud computing.

B. Why Use Unikernels?

Using HW virtualization as the preferred isolation mechanism requires an operating system to be embedded into the virtual machine. IaaS cloud providers will typically offer virtual machine images running a classic general purpose operating system, such as Microsoft Windows and one or more flavours of Linux, possibly optimized for cloud by, e.g., removing device drivers that are not needed. While specialized Linux distributions can greatly reduce the memory footprint and

software attack surface of a virtual machine, general purpose multi-process operating systems will, by design, contain a large amount of functionality that is simply not needed by one single service. We take the position that virtual machines should be specialized to a high degree, each forming a single purpose micro service, to facilitate a resilient and fault tolerant system architecture, which is also highly scalable.

We argue that the unikernel approach offers potential to meet all our needs, while delivering a much reduced software attack surface, yet providing exactly the performance we require. An added bonus will be the reduced operating footprint, meaning a more green approach is delivered at the same time.

C. How Does This Compare to a Conventional System?

Looking at what Frederick P. Brooks Jnr. suggests in [26] "Because ease of use is the purpose, this ratio of function to conceptual complexity is the ultimate test of system design. Neither function nor simplicity alone defines a good design", we can see where modern software systems are missing the point. The more complex a system becomes, the more overhead is introduced, leading to greater complexity and unnecessary bloat, draining performance, and exposing vulnerabilities. Conventional cloud systems tend to be over-complicated, unnecessarily bloated, and thus expensive to scale. Unikernels, in [6], "Unikernels are specialized, single-address-space machine images constructed by using library operating systems", meaning they are exactly the right size to carry out their given task — no larger, and no smaller.

Our approach, using unikernels, limits/enforces the software architect to use a given pattern (event-based computing using the single-responsibility-principle, service-oriented architectures, separation of data and processing, and modularity) — which is very good from a software design point of view. We are trying to get people to use "best-of-breed" patterns, and thus develop better software through this limitation.

IX. How Does This Address Our Ten Key Concerns?

As we saw in the introduction, we identified 10 key security issues needing to be addressed. We believe our unikernel solution can help us address seven of these issues, namely: The definition of security goals; Compliance with standards; Audit issues; Management approach; Technical complexity of cloud; Measurement and monitoring; The threat environment.

A. The Definition of Security Goals

By design, we will build in a number of sensible security goals to the system. We can also accommodate additional goals, where the user identifies those as appropriate.

B. Compliance with Standards

Compliance is generally achieved through some form of assurance [27], which generally can be achieved by a compliance process or by audit. Audit is expensive if done well, thus compliance through the use of checklists is the usual method chosen, but brings weaknesses with it [28]. Tightening information flows within the system, and providing rigorous audit trails, maximises assurance, leading to compliance in a much more accurate and cost effective way.

C. Audit Issues

Many audit issues needing to be addressed [29], especially those surrounding the use of the humble audit trail [30]. In a forthcoming paper, we outline in more detail how our system will tackle this key issue with a much more rigorous approach.

D. Management Approach

Cloud ecosystems involve far more actors than conventional systems, and many of these actors have differing agendas [31]. Our approach seeks to minimise the impact of third party actors by reducing the opportunity for these actors to adversely influence the effectiveness of the security approach.

E. Technical Complexity of Cloud

Distributed systems are highly complex. Cloud ecosystems are, by their nature, far more complex [32]. We propose to tackle this issue through simplification of the system architecture, to minimise the software attack surface.

F. Measurement and monitoring

To achieve a provable level of security [33], it is necessary to measure and monitor what is happening with a system. Our system will, by default, provide a considerable armoury of measurement and monitoring capabilities, which will allow users to be satisfied of the level of security they have achieved, and will continue to achieve through continuous monitoring.

G. The threat environment

This is a major and very worrying issue, which continues to evolve day by day. Our approach seeks to tackle this through minimising software attack surface, minimising access routes to attackers, and generally making life difficult for the attackers. This area will need ongoing scrutiny by the research community in order to try to keep ahead of the attackers.

X. Initial Thoughts on Penetration Testing

Penetration testers often refer to the OWASP foundation Top 10 report, see Table II below for details of the most used attack techniques. In its current 2013 installation, two vulnerabilities—A5-Security Misconfiguration and A9-Using Known Vulnerable Components—are directly related to the rich landscape of available server-side functions, which commonly are neither minimized nor properly configured. Recent years have given rise to opinionated frameworks, i.e., frameworks that guide developers with sensible security defaults. Their security measures efficiently reduce threats from common attack vectors, e.g., A1-Injection or A2-Cross-Site Scripting, but those frameworks themselves can introduce vulnerabilities, as OWASP noted with its introduction of A9 as "the growth and depth of component based development has significantly increased the risk of using known vulnerable components". The Unikernel approach implicitly minimizes infrastructure — runtime environment, and libraries as well as operating system shells — and thus reduces exposure to attack vectors A5 and A9. Plus, their single-process paradigm

 ISBN: 978-1-61208-529-6

enforces beneficial architecture design decisions, yielding systems with clearer separation-of-concerns. Given the rise of opinionated frameworks, we envision a web-

TABLE II. OWASP TOP TEN WEB VULNERABILITIES — 2013 [34]

2013 Code	Threat
A1	Injection Attacks
A2	Broken Authentication and Session Management
A3	Cross Site Scripting (XSS)
A4	Insecure Direct Object References
A5	Security Misconfiguration
A6	Sensitive Data Exposure
A7	Missing Function Level Access Control
A8	Cross Site Request Forgery (CSRF)
A9	Using Components with Known Vulnerabilities
A10	Unvalidated Redirects and Forwards

development framework that de-constructs high-level workflows into separate unikernels, structures communication between those, and provides sensible security defaults. We assume that such a system of unikernels can solve complex web-application workflows in a secure manner without negatively impacting developer's productivity during development and debugging, which we next address in much more depth.

XI. Conclusions

We introduced a framework of definitions and metrics for classifying unikernel systems, and began developing a formal approach to describing our framework, and considered how such a theoretical framework might provide a more secure approach to the challenges of cloud security and privacy.

We have proposed a novel means of significantly reducing the software attack surface for a cloud based system, removing in the process many classic attack vectors. We consider the architecture of the proposed system and its resilience to attack in much more depth in our forthcoming publications.

We need to look at, and solve, the challenge presented by audit trail issues, which will require secure internal communication, access logging and log storage, and provision of a strong forensic trail. Once these security basics are in place, we can turn our attention to a robust approach to privacy.

References

[1] B. Duncan, A. Bratterud, and A. Happe, "Enhancing Cloud Security and Privacy: Time for a New Approach?" in *INTECH 2016*, Dublin, 2016, pp. 1–6.

[2] B. Duncan, A. Happe, and A. Bratterud, Enterprise IoT Security and Scalability: How Unikernels can Improve the Status Quo, in *9th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2016)*, 2016, Shanghai. pp. 16.

[3] A. Madhavapeddy, et al., "Unikernels: Library Operating Systems for the Cloud," *ASPLOS '13 Proc. eighteenth Int. Conf. Archit. Supp Prog. Lang. Oper. Syst.*, vol. 48, pp. 461–472, 2013.

[4] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the Virtual Library Operating System," *Commun. ACM*, vol. 57, no. 1, pp. 61–69, 2014.

[5] A. Kantee, "The Rise and Fall of the Operating System," *Login:*, pp. 6–9, 2015.

[6] Unikernel.org, "Unikernels," 2015. [Online]. Available: www.unikernel.org Last accessed: 11 Jan 2017

[7] D. R. Engler, M. F. Kaashoek, and Others, *Exokernel: An operating system architecture for application-level resource management*. ACM, 1995, vol. 29, no. 5.

[8] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," in *ACM SIGPLAN Not.*, vol. 46, no. 3. ACM, 2011, pp. 291–304.

[9] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, p. 8, 2015.

[10] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *ACM SIGOPS Oper. Syst. Rev.*, vol. 7, no. 4, p. 112, 1973.

[11] M. F. Kaashoek, et al., "Application Performance and Flexibility on Exokernel Systems," *ACM SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 52–65, 1997.

[12] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services," *2015 IEEE 7th Int. Conf. Cloud Comput. Technol. Sci.*, pp. 250–257, 2015.

[13] G. Klein, et al., "seL4: Formal verification of an OS kernel," *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ.*, pp. 207–220, 2009.

[14] A. Bratterud and H. Haugerud, "Maximizing Hypervisor Scalability Using Minimal Virtual Machines," *2013 IEEE 5th Int. Conf. Cloud Comput. Technol. Sci.*, pp. 218–223, 2013.

[15] P. Meireles, "Narkive Mailinglist Archive," 2007. [Online]. Available: http://m0n0wall.m0n0.narkive.com/OI4NbHQq/m0n0wall-virtualization Last accessed: 05 Jan 2017.

[16] PWC, "Information Security Breaches Survey 2010 Technical Report," pp. 1–22, 2010.

[17] PWC, "UK Information Security Breaches Survey - Technical Report 2012," London, Tech. Rep. April, 2012.

[18] PWC, "2014 Information Security Breaches Survey: Technical Report," Tech. Rep., 2014.

[19] Verizon, N. High, T. Crime, I. Reporting, and I. S. Service, "2012 Data Breach Investigations Report," Verizon, Tech. Rep., 2012.

[20] Verizon, "2014 Data Breach Investigations Report," Tech. Rep. 1, 2014.

[21] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[22] P. Barham, et al., "Xen and the art of virtualization." *SIGOPS Oper.Syst.Rev.*, vol. 37, no. 5, pp. 164–177, 2003.

[23] D. Chisnall, "Xen PVH: Bringing Hardware to Paravirtualization." *Inf. IT*, 2014.

[24] X. Project, "Xen Project Software Overview," 2016. [Online]. Available: http://wiki.xen.org/wiki/Xen_Project_Software_Overview#PVH Last accessed: 05 Jan 2017.

[25] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," *Proc. Tenth Eur. Conf. Comput. Syst. - EuroSys '15*, pp. 1–17, 2015.

[26] F. P. Brooks Jr, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.

[27] B. Duncan and M. Whittington, "Compliance with Standards, Assurance and Audit: Does this Equal Security?" in *Proc. 7th Int. Conf. Secur. Inf. Networks*. Glasgow: ACM, 2014, pp. 77–84.

[28] B. Duncan and M. Whittington, "Reflecting on whether checklists can tick the box for cloud security," in *Proc. Int. Conf. Cloud Comput. Technol. Sci. CloudCom*, vol. 2015-Febru, no. February. Singapore: IEEE, 2015, pp. 805–810.

[29] B. Duncan and M. Whittington, "Enhancing Cloud Security and Privacy: The Cloud Audit Problem," in *Cloud Comput. 2016 Seventh Int. Conf. Cloud Comput. GRIDs, Virtualization*. Rome: IEEE, 2016, pp. 119–124.

[30] B. Duncan and M. Whittington, "Enhancing Cloud Security and Privacy: The Power and the Weakness of the Audit Trail," in *Cloud Comput. 2016 Seventh Int. Conf. Cloud Comput. GRIDs, Virtualization*. Rome: IEEE, 2016, pp. 125–130.

[31] B. Duncan and M. Whittington, "Company Management Approaches Stewardship or Agency: Which Promotes Better Security in Cloud Ecosystems?" in *Cloud Comput. 2015*. Nice: IEEE, 2015, pp. 154–159.

[32] B. Duncan, D. J. Pym, and M. Whittington, "Developing a Conceptual Framework for Cloud Security Assurance," in *Cloud Comp. Tech. Sci. (CloudCom), 2013 IEEE 5th Int. Conf. (Vol 2)*. Bristol: IEEE, 2013, pp. 120–125.

[33] B. Duncan and M. Whittington, "The Importance of Proper Measurement for a Cloud Security Assurance Model," in *2015 IEEE 7th Int. Conf. Cloud Comput. Technol. Sci.*, Vancouver, 2015, pp. 1–6.

[34] OWASP, "OWASP Top Ten Vulnerabilities 2013," 2013. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project Last accessed: 05 Jan 2017.

 ISBN: 978-1-61208-529-6