Unikernels for Cloud Architectures: How Single Responsibility can Reduce Complexity, Thus Improving Enterprise Cloud Security

Andreas Happe[1], Bob Duncan[2] and Alfred Bratterud[3]
[1]*Dept. Digital Safety & Security, Austrian Inst. of Tech. GmbH, Vienna, Austria*
[2]*Computing Science, University of Aberdeen, Aberdeen, U.K.*
[3]*Dept. of Computer Science, Oslo and Akershus University, Oslo, Norway*

Keywords: Cloud Security and Privacy, Attack Surface, Compliance, Complexity.

Abstract: Unikernels allow application deployment through custom-built minimal virtual machines. The authors investigate how unikernels and their inherent minimalism benefit system security. The analysis starts with common security vulnerability classes and their possible remediation. A platonic unikernel framework is used to describe how unikernels can solve common security problems, focusing both on a micro- and macro level. This theoretical framework is matched against an existing unikernel framework, and the resulting mismatch is used as a starting point for the research areas the authors are currently working on. We demonstrate how using a single responsibility unikernel- based architectural framework could be used to reduce complexity and thus improve enterprise cloud security.

1 INTRODUCTION

There is a commonly held view in security circles that software complexity is the enemy of security. Virtualisation is often employed to split up applications into manageable isolated parts. Figure 1 gives a rough overview about different virtualization techniques. Unikernels combine lightweight virtualisation while placing limitations upon potential application code. These limitations yield solutions that heed the single responsibility principle, which, by reducing complexity, ensures security is more easily achieved.

In previous work [Duncan et al., 2016a], we outlined how a new approach to cloud security and privacy might provide a better solution to the challenges of good cloud security and privacy. In that work, we identified 10 key management challenges, and suggested how a unikernel could assist in addressing 7 of these challenges to ensure good cloud security and privacy can be achieved. The defining theme was that complexity is the natural enemy of security; unikernel systems with their enforced minimalism can help prevent common security problems. In [Bratterud et al., 2017], we looked at the technical solution in much more depth, defining and demonstrating how unikernels work. We took a detailed look at how the approach is likely to work in the real world, considering how unikernels fit into a larger software project, what capabilities are currently lacking and how those can be incrementally added to a unikernel framework.

Our contributions for this paper are to map the OWASP Top 10 towards idealized unikernel environments; to compare unikernels to serverless architectures; to identify unikernel's shortcomings in development environments; and to lay the groundwork for a multi-unikernel framework.

We base this work on empirical security findings from real-world security breaches. We start by discussing currently exploited vulnerabilities for web applications in Section 2, analyzing which vulnerabilities will be prevented by the use of unikernels in Section 3. Unikernels are part of an overall software architecture, and in Section 4 we examine patterns for that architecture that can aid secure software development. In Section 5, we examine how an existing unikernel framework—UniK—fits our proposed ideal framework. Following this, in Section 6 we clarify how a future framework might improve on the status quo. In Section 7, we discuss our conclusions, currently performed research and potential future steps.

Happe, A., Duncan, B. and Bratterud, A.
Unikernels for Cloud Architectures: How Single Responsibility can Reduce Complexity, Thus Improving Enterprise Cloud Security.
DOI: 10.5220/0006282800300041
In *Proceedings of the 2nd International Conference on Complexity, Future Information Systems and Risk (COMPLEXIS 2017)*, pages 30-41
ISBN: 978-989-758-244-8

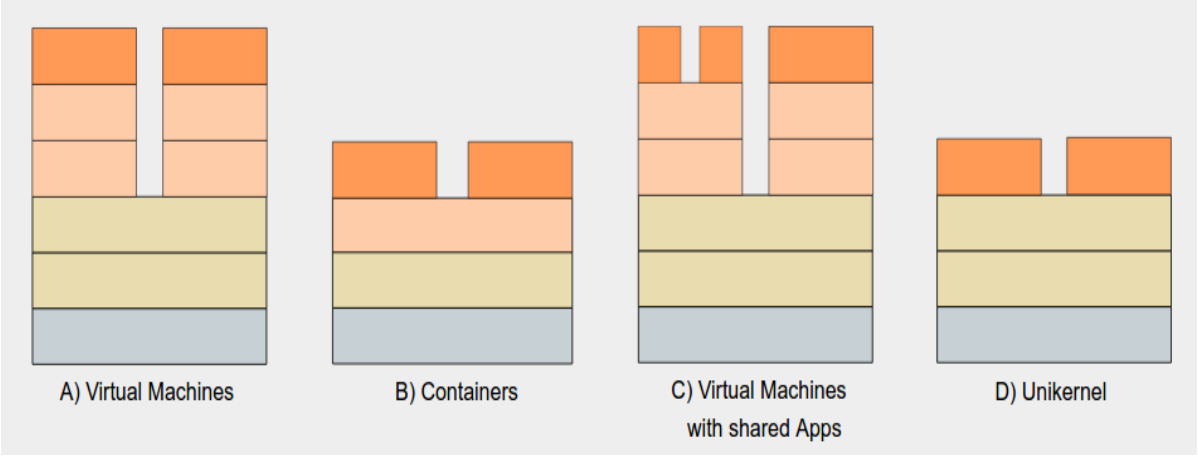

Figure 1: A) Traditional virtual machines offer the highest level of service isolation but introduce high overhead as each application should—theoretically—be placed within a virtual machine of its own. B) Containers improve efficiency by reusing large parts of the host operating system and kernel, but are thus not able to provide the security guarantees of fully separated virtual machines, e.g., resource starvation as well as kernel-level exploits can have cross-container impact. C) A new trend is to encapsulate one or more containers into virtual machines on their own. This improves security and service isolation but introduces the overhead of virtual machine based solutions. D) Unikernels, in contrast, are minimal virtual machines thus yield the high security assumptions of virtual machine based approaches while improving efficiency through their minimalism.

2 THE OWASP TOP 10

Security breaches have a negative monetary and publicity impact on companies, thus are seldom publicly reported. This limits the availability of empirical study data on actively exploited vulnerabilities. The OWASP Foundation Top 10 report [OWASP, 2013], is a rare exception: it provides a periodic list of exploited web-application vulnerabilities, ordered by their prevalence. OWASP focuses on deliberate attacks, each of which might be based on an underlying programming error, e.g., an injection vulnerability might be the symptom of an underlying buffer-overflow programming error.

The current 2013 report lists these vulnerabilities:

Table 1: OWASP Top Ten Web Vulnerabilities — 2013 [OWASP, 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

Based on software development impact, different vulnerabilities can be grouped into three classes:

1. low-level vulnerabilities can be solved by applying local defensive measures, e.g., using a library at a vulnerable spot;
2. high-level vulnerabilities cannot be solved by local changes but need systematic architectural treatment; and
3. application specific vulnerabilities cannot be solved in a generic manner but depend on thoughtful developer intervention.

Two of the top three vulnerabilities (A1, A3) can be categorized as “low-level” and are directly related to either missing input validation or output sanitization, which can be mitigated by consistently using defensive security libraries. Another class of attacks that can similarly be solved is A8. In contrast, “high-level” vulnerabilities must be solved on an architectural level. Examples are A2, A5 and A7. The software architecture should provide generic means for user authentication and authorization, and should enforce validations for all operations. Examples for the final class can be seen in A4, A6 and A10: these directly depend on application workflow and can only be solved through application-dependent changes.

Some vulnerabilities can be prevented by consistently using security libraries while others can be reduced by enforcing architectural decisions during software development. Recent years have given rise to opinionated software development frameworks,

e.g., Ruby on Rails [37signals,]. Those frameworks guide software development by providing a sensible collection of support libraries (including defensive security libraries) as well as strongly favouring a given architectural programming style, e.g., Ruby on Rails favouring the MVC pattern [Burbeck, 1992], in combination with the ActiveRecord data-access pattern [Fowler, 2002]. Software security is commonly a non-functional requirement and thus hard to get funding for. Opinionated frameworks allow software developers to focus on functional requirements while taking care of many security implications.

Those security frameworks have grown in size and functionality, and being software themselves, can introduce additional security problems into otherwise secure application code. While the Ruby on Rails framework, when properly used, prevents many occurrences of XSS-, SQLi- and CSRF-Attacks, problems with network object serialization recently introduced remotely exploitable injection attacks [Climate, 2013]. This capability was not commonly used, but was included in every Ruby on Rails installation. Similar problems have plagued Python and its Django framework [Blankstein and Freedman, 2014]. This is further aggravated as software frameworks are by design generic: they introduce additional software dependencies not used by the application code at all. Their configuration often focuses on usability including easy debug infrastructure—from a security perspective everything that aids debugging aids penetration. In its 2013 report OWASP acknowledged this problem by introducing A9, to address "the growth and depth of component based development has significantly increased the risk of using known vulnerable components" [OWASP, 2013].

Another recent security battleground is the Internet-of-Things (IoT). Recent examples are the 1TB/sec attack against "Krebs on Security" or the DDoS Attack taking out Dyn in October 2016[1]. While the generic security landscape is already complex, IoT adds additional problems such as hard device capability restrictions, manifold communication paths and very limited means of updating already deployed systems. In addition, software is often an after-thought. This leads to a situation where security updates are scarce in the best scenario.

3 UNIKERNELS

While Madhavapeddy provides an in-depth review of unikernels in [Madhavapeddy and Scott, 2013], [Madhavapeddy et al., 2013], we can use a simpler definition. From a security perspective, the most intriguing aspect of unikernels is their capability as a minimal execution environment. We view unikernels as black-boxes providing the following attributes:

- a minimal execution environment for a service;
- provides isolation between different services;
- the unikernel image is immutable, i.e., no data alteration can be persisted within the unikernel itself;
- a single-execution flow in a single-process namespace, i.e., neither multi-tasking nor multi-threading. The unikernel is the synthesis of an operating system kernel and the application.

Cloud computing consolidates processing power within data centres to achieve maximum utilization and high energy efficiency. Multiple applications run on the same computing node. Control on node placement or concurrently running applications is seldom possible, making isolation between different applications, users, or services, critical for security. A common but inefficient solution is to place each application or service within a virtual machine [Jithin and Chandran, 2014]. This is very similar to the initial usage of virtualization within host-based systems; Madnick gives a good overview of the impact of virtualization in [Madnick and Donovan, 1973]. Containers offer a more efficient approach [Soltesz et al., 2007], but originally developed to improve deployment, their security benefits are still under debate [Bui, 2015].

Go programs [Pike, 2009], are statically compiled and linked, thus creating a single binary including all needed dependencies for execution. While this is conceptually similar to unikernels, Go programs still depend on a shared operating system kernel—impacting negatively on service isolation and security.

Minimization is performed during unikernel building, meaning the system image only includes required software dependencies: implying no binaries, shell or unused libraries are included within the unikernel image. Further, even unused parts of libraries should not be included in the image at all. This radically reduces included functionality and thus — if that functionality was network-accessible — reduces the network attack surface tremendously[2]. Also this can loosen the need for updates after vulnerabilities have been discovered in included third-party

[1] https://www.theguardian.com/technology/2016/oct/25/ddos-cyber-attack-dyn-internet-of-things

[2] We use OWASP's definition of attack surface, which includes the sum of all paths for data/commands into and out of the applications as well as all valuable data used by the application. In addition, all protection mechanisms are included within the attack surface [OWASP,].

components — if the vulnerable function was not included within the set of used functions, an update can be moot. A similar theme can be found with the ongoing Gnome Flatpak [Flatpak,], and Ubuntu Snap [Ubuntu,], projects: both provide packaging for application containers. Flatpak can police the allowed access to system resources (processes, file system) from within the application container, but by their own definition, is not intended to be used for servers.

Another benefit of the applied minimalism is a reduced memory footprint [Bratterud and Haugerud, 2013], [Bratterud et al., 2015], and quick start-up time of unikernel-based systems. Madhavapeddy et al., used this for on-demand spin-up of new virtual images [Madhavapeddy et al., 2015], allowing for higher resource utilization and improved energy efficiency.

Minimalism also benefits next-generation hardware-supported memory protection techniques. Intel Secure Guard Extensions [Anati et al., 2013, Costan and Devadas,], allow for protected memory enclaves. Direct access to those enclaves is prohibited and protected through specialized CPU instructions; as the protection is enforced by hardware, even the hypervisor can be constrained from accessing protected memory. Rutkowska has shown [Rutkowska, 2013], that deploying this protection scheme for applications has severe implications: just protecting the application executable is insufficient as attacks can inject or extract code within linked libraries. This leads to the conclusion that the whole application including its dependencies must be part of the secure-memory enclave. Simplicity leads to a "one virtual machine per application" model. Unikernels inherently support a single application per unikernel. We propose unikernels are a perfect fit for usage with advanced memory protection techniques.

With unikernels, the situation after a security breach vastly differs to traditional systems. The attacker is able to exploit a vulnerability, e.g., buffer overflow, gaining access to the unikernel system's memory. Having no binaries and reduced libraries, writing shell [Arce, 2004], code is complicated[3]. Pivot attacks that depend on shell-access are thwarted. On the other hand, all direct attacks against the application, e.g., data extraction due to insecure application logic, are still possible. A good example is the recent OpenSSL heartbleed vulnerability [Durumeric et al., 2014]. A unikernel utilizing OpenSSL would also be vulnerable, allowing an attacker to access system memory, including the private SSL key. We argue that functionality should be split between multiple unikernels, compartmentalizing breaches.

The lack of shell access requires mental readjustment for many UNIX-period system administrators. On the other hand, the growing DevOps movement [Bass et al., 2015], abolishes the traditional separation into software development and system administration, but places high importance on the communication between and integration of those two areas. Unikernels offer an elegant deployment alternative: the minimized operating system implicitly moves system debugging to application developers. Instead of analysing errors through shell commands, developers can utilize debuggers to analyse the whole system, which might be beneficial for full-stack engineering.

Returning to the theme of "*software development frameworks provide sensible defaults but get bloated, and thus vulnerable over time*", unikernels provide an elegant solution: while the framework should include generic defence measures, the resulting unikernel will only include needed parts to reduce the attack surface.

3.1 Influence of Language Selection

In contrast to general purpose operating systems, most unikernels are written with a target programming language in mind.[4] As the application code will be implemented with this language, the language selection has enormous impact on application security.

An especially error-prone area is memory-safety and memory management. Flexible programming languages such as C [Kernighan et al., 1988], or C++ [Stroustrup, 2015], allow direct manipulation of memory areas, e.g., pointer arithmetic, casting and manual memory allocation. While necessary for high-performance applications this introduces various potential security problems such as buffer overflows, dangling pointers, use-after-free, typecast errors, etc. Many high-level languages disallow pointer arithmetic and exchange manual memory management with a garbage collector.

Strongly-typed languages allow detection of common programming errors during compile time, preventing vulnerabilities that could potentially be ex-

[3]Shellcode is machine code that is used as a payload during vulnerability execution. Common payloads spawn command shells or abuse existing libraries to give attackers unintended possibilities.

[4]Please note, that the unikernel itself might be written in multiple different programming languages. Memory-safe and type-safe languages lack the flexibility of raw memory access/type-casting and thus are too limited for kernel implementation [Duncan et al., 2016b]. This commonly yields a combination of unsafe languages for low-level memory interactions and safe high-level languages interfacing the application code, for example, Microsoft's Verve [Yang and Hawblitzel, 2010], combines C++ and managed C#, HalVM uses C/Assembler and Haskell, MirageOS is based on C and OCaml.

ploited by runtime attacks, thus a strongly typed language is preferable. Determining the strong-typedness of a language is not easy. Strongly typed languages such as C or C++ allow unsafe explicit type conversions. Languages such as Java are based on *type erasure* [Bracha et al., 1998]: they perform type checking during compilation but remove details during run-time, thus preventing dynamic type checking.

The selected programming language should provide usable abstractions suitable for the intended field of operation. If object-oriented programming is intended, then the unikernel's programming language should support this style. Furthermore the language should provide means of programming fluently within the chosen paradigm, e.g., for object-oriented programming styles SOLID [Martin, 1995], principles should be supported, as SOLID's principles are also important for security. SOLID's S — Single Responsibility Principle — and I — Interface Segregation Principle — yields software designs that depend on multiple distinct objects, each offering minimal but self-contained functionality. This resonates with the unikernel paradigm.

Another important aspect for security efficiency is tooling and library support. Recently we have seen programming languages providing good tooling enjoying better uptake by the community [Meyerovich and Rabkin, 2013]. Such an example where uptake was improved by tooling is Ruby with the Ruby on Rails web-development framework. The Go programming language goes a step further and includes its own package management system out of the box. Library support can be security-relevant, e.g., the C language does not provide for memory-safety with respect to memory management but add-on libraries that introduce memory-safety [Bhatkar et al., 2005], [Cowan et al., 1998], or automated garbage collection [Detlefs et al., 1994], are available.

To summarize the impact on language selection for unikernels:

- must support the targeted programming paradigm;
- should support usable software-development abstractions;
- strongly-typed and memory-safe programming languages yield security benefits;
- good tooling support (e.g., safety checkers, static code analysis) can improve security and developer efficiency.

3.2 Unikernels and Serverless Stacks

Another recent addition to the hosting repertoire are serverless frameworks. With those frameworks, source code is directly uploaded to the cloud service. Execution is triggered in response to events; resources are automatically scaled. Developers do not have any system access except through the programming language and provided libraries. Most current offerings for this software stack are highly vendor-dependent, e.g., Amazon Lambda, Google Cloud Functions, Microsoft Azure Functions.

We do see unikernel and serverless frameworks as two solutions to a very similar problem: reducing the administrative overhead and allowing developers to focus their energy on application development. Serverless stacks signify the "corporate-cloud" aspect: developers upload their code to external services and thus invoke vendor lock-in in the long run. Unikernels also allow users to minimize the non-application code; in contrast to serverless architectures, this approach maintains flexibility with regard to hosting. Users can provide on-site hosting or move towards third-party cloud offerings.

We expect serverless architecture providers to utilize unikernels within their own offerings. They are well suited to include user provided applications and further increase security of their infrastructure.

3.3 Influence on Architecture

High-level vulnerabilities cannot be solved on a low-level but must be re-mediated through architectural decisions. As mentioned, we entertain a black-box view of a unikernel as an immutable isolated single-execution-flow execution environment. This influences both the internal architecture of applications running within unikernels as well as the overall architectural structure of software systems.

The single-execution-flow approach leads naturally to a reactive [Bonér et al., 2014], event-driven architecture [Fielding, 2000]. In such an architecture, the application sequentially retrieves and fulfills user requests. There is emphasis on the non-blocking processing of incoming requests[5]. If no unprocessed request is available, the application enters a sleeping state, thus preserving system resources. No two requests are processed at the same time by the same unikernel process—this prevents race conditions, as there can be no shared data. In addition, there is efficient operating-system support for event-loops, allowing for resource efficient implementations, thus

[5]This is in contrast to traditional procedural processing where blocking function calls are common. While blocking, the caller waits for an external entity to answer, which implies network-communication or waiting on an additional process. The latter is not allowed with our unikernel-approach and the former is bad for efficiency.

reducing energy consumption. If parallel processing is mandatory, an additional unikernel running the same operation can be spawned. We assume that no state or data is shared between those unikernels, i.e., they can perfectly run in parallel without any locking up on shared data. A prominent example of an event-driven architecture is the nginx web server.

Unikernel images are immutable. This implies that all data alteration operations cannot be executed within the unikernel, but must be delegated to an external service. While initially sounding like a limiting factor this actually is good architectural practice as it separates data processing from data storage. Please note, that unikernels are not side-effect free (also called "pure" within functional programming languages [Wadler, 1992]) — while the data itself is immutable, side-effects can be introduced, e.g., through usage of a random number generator.

The combination of immutable unikernel images and event-driven execution allows for easier reasoning about the overall software system. Security-wise complexity is the natural enemy of good security, thus this change is highly appreciated. Performance-wise this combination should allow for good scale-out behaviour, i.e., additional unikernels can be spawned if increased processing throughput is needed. This highly resembled the Lambda Architecture [Marz and Warren, 2015], [Fan and Bifet, 2013], commonly used for data warehousing. On the downside, a purely event-driven programming style with a single-execution-flow is not well-suited for interactive user interfaces that drive application logic: here input processing—and thus the user interface—would stall until the submitted operation is completed.

We assume this software architecture will primarily be used for network-driven request processing. Immutability moves the need for data synchronization outside the unikernel: the overall framework must provide some means of persistence for shared data. However, this implementation must be able to cope with concurrency, i.e., provide ACID[6] guarantees or deal with the CAP theorem[7]. Note, that "shared data" includes audit data, session data, perhaps even caches.

As shown, the natural evolutionary step would be separation of operations into multiple unikernels. This allows for better security compartmentalization as well as positively enabling scale-out operations. Creating the communication infrastructure between those unikernels, plus performance monitoring, must again be provided by the surrounding framework.

[6]ACID describes a set of properties (Atomicity, Consistency, Isolation and Durability) desirable for data transactions [Haerder and Reuter, 1983].

[7]The CAP theorem describes the non-reconcile-ability between Consistency, Availability and Partition-Tolerance — a maximum of two of those three desirable properties can be provided [Abadi, 2012].

4 A FRAMEWORK FOR UNIKERNELS

The generic scenario for our framework is providing network-connected services through an immutable single-execution-flow unikernel. We limit ourselves to single-execution-flow kernels to optionally allow for parallelism optimizations within multiple unikernel systems later. The immutability assumption minimizes requirements for targeted unikernels. The network interfaces are supposed to be accessible through HTTP over TCP/TLS. We chose HTTP due to its ubiquitous usage within internet applications. We assume communication between unikernels will be message based. This will be further addressed in our work on unikernel frameworks, in which we investigate whether this would be the right approach.

Figure 2 shows a realization of such a framework. The separation of the responsibility principle led to the split into multiple purpose-built unikernels. A quick description of the request flow shows the different unikernels' responsibilities. The initial *TLS terminator* accepts incoming client SSL/TLS connections. Its implementation as a separate unikernel shows both advantages and problems of unikernel-based designs: the TLS/SSL-endpoint inherently needs access to the SSL private key. Using the unikernel approach allows us to only include this key within the TLS unikernel and exclude it from all other runtime processes, thus greatly reducing the system's attack surface. On the downside, the unikernel-approach will not protect against a vulnerable SSL implementation (think OpenSSL heartbleed bug [Durumeric et al., 2014]) — in contrast, due to the reduced memory size, a heartbleed attack would be performed more efficiently due to less memory copying needed. After TLS termination the request is passed to a transparent *HTTP Proxy* which caches incoming requests based on their HTTP headers. If the request is fresh, the *Router* unikernel forwards the request to the matching unikernel performing the corresponding operation. During operation execution, the unikernel may interact with framework-provided components: it may access persistent data stores or provide log entries to the audit system.[8] The operation's response is then transformed by the *Encoder* into the client-specified

[8]A full unikernel authentication and messaging solution is currently under development by the authors.

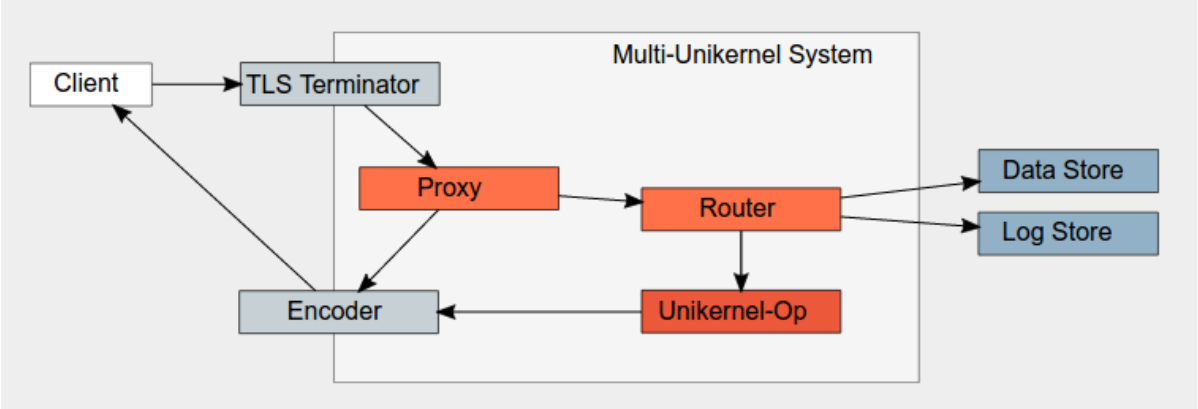

Figure 2: Multi-Unikernel System consisting of connected microservices and external components: Clients connect to the system through the incoming TLS-Connector and outgoing Encoder component – both contain state and thus are not implemented as unikernels. Framework-provided components, such as the transparent proxy or message router, are implemented as functional unikernels if possible. Application functionality is provided through user-supplied unikernels (red) which may depend on external stateful services (blue).©2016 Happe, Duncan and Bratterud.

representation — e.g., "application/xml", "application/json" or "text/html" — and then returned to the original client. In case of generated HTML files we envision the usage of templating engines. Additional administrative services are provided by the framework, including unikernel monitoring and updating.

Please note, that we have not included countermeasures against OWSAP Top 10's top vulnerability: injection attacks. The commonly accepted way of dealing with this attack vector is consistent input sanitization of all provided user input. We can provide this within a unikernel framework on multiple levels: on a coarse level a new "anti-injection" unikernel employing a generic sanitization library which can be placed between the *HTTP Proxy* and *Router* unikernel, thus transparently filtering the whole incoming traffic. Sometimes filtering the whole traffic is not feasible due to performance constraints. In this case, the filtering unikernel might be placed directly between the *Router* and selected *Operation* unikernels, thus only filtering a fraction of the original traffic.

4.1 High-level Concerns

The main purpose of the framework is the deconstruction of business workflows into unikernel-based operations. A single workflow can be implemented through multiple chained unikernels. The surrounding framework is responsible for creation, monitoring and stopping the different unikernel-services during runtime. While unikernels themselves provide good functional service isolation, external monitoring is needed to prevent starvation attacks, i.e., one unikernel performing a denial-of-service attack by consuming all available host resources.

The framework must take care of creating communication channels between connected unikernels. It should provide means of validating passed messages, i.e., deep-packet inspection. This will be utilized to further compartmentalize potential security breaches.

Real-world use-cases require mutable data, e.g., temporary states, logging information or persisted application and or user data. Unikernels are by definition immutable. To solve this mismatch, the surrounding framework must provide means for persisting and querying data in a race-free manner. Specialized data storage might be provided depending on the use case. For example, log and audit data's special access patterns warrant special storage procedures enforcing its append-only nature. Persistent data-storage is inherently contrary to our immutable unikernel approach. Being pragmatic, we do not enforce data storage within unikernels but defer this functionality to the enclosing framework, i.e., means of storage are provided by the environment.

The resulting framework fulfills all requirements of *Crash-Only Software* described first by Candea and Fox[Candea and Fox, 2003] during the 9th Usenix Workshop on Hot Topics in Operating Systems. Crash-only software can safely be restarted in case of errors. To allow for this, multiple requirements have to be fulfilled:

1. *All important non-volatile state is managed by dedicated state stores*;
2. *Components have externally enforced boundaries*;
3. *Interactions between components have timeouts*;
4. *All resources are leased*;
5. *Requests are entirely self-describing*.

Unikernel-based frameworks with their immutable unikernel images automatically fulfill requirements 1 and 2. This implicitly also fulfills requirement 5. The missing requirements (3 and 4) must be implemented and fulfilled by the corresponding framework's environment.

4.2 Message- vs. Stream-based Communication

In our proposed platonic framework we have chosen a message-based communication architecture. On a fundamental level an incoming HTTP request uses both stream-based as well as message-based communication mechanisms: each incoming HTTP message is encapsulated within a TCP stream. The initial HTTP/HTTPS endpoint component is responsible for transforming the encapsulated messages into atomic HTTP messages. Communication between components is then based on a message-based approach. This allows for easy storage of in-flight messages. This benefits hot-replacement of unikernels: after a unikernel has been restarted—or replaced—the incoming message queue is triggered to replay all stored messages and thus ensures that all incoming operations are executed. The disadvantage of message-based processing is the inherent memory overhead. As all messages need to be stored within memory this architecture is not well suited for applications that process "large" requests, such as large file delivery.

4.3 Resilience and Zero-Downtime-Updates

In addition to security benefits, the combination of immutable single-execution-flow unikernels also allows for improved availability. An essential step during software lifetime is updates: a new version of the software has to be deployed "over" an existing deployment, data migrated to the new version should happen without any impact on service availability.

The combination of immutable unikernel, external data storage and the existence of an external unikernel monitoring and management framework allows for exactly that. When a new version of an operation is available, a new unikernel is compiled and deployed. Incoming messages for the old unikernel are paused, the framework waits until the old unikernel has finished processing. Afterwards, incoming messages are forwarded to the new unikernel and message processing is resumed. If database alterations are needed, those are performed after the old unikernel has stopped and the new unikernel is receiving messages. Due to the immutable nature of unikernels this hot-replacement of unikernels is possible.

Special consideration has to be given for the initial *TLS-Terminator* unikernel: TCP is a stream-based protocol, the initial unikernel has to keep track of the SSL/TLS-state within its volatile memory. This prevents simple transparent replacement operations, but due to the minimal provided functionality we assume replace operations of this unikernel to be infrequent.

We assume the software system comprises multiple unikernels, i.e., one unikernel per offered operation, allowing us to replace defective parts of the software system without introducing overall downtime.

5 CURRENT STATE: UNIK

We hold the benefits of a framework centred around unikernel creation and deployment to be self-evident. This begs the question, if such a framework already exists, or if not, which capabilities are currently missing to achieve the desired functionality.

EMC's UniK is the the prime unikernel compilation and deployment platform. This characterization comes easy, as it also is the only solution that supports multiple unikernel implementations, e.g., rumprun, IncludeOS, MirageOS, and OSv, as well as multiple deployment backends—AWS/Xen, VirtualBox, Qemu/KVM, vSphere and Photon Controller[9]. Building unikernels into virtual machine images is internally delegated to the corresponding toolkits, UniK manages those virtual machine images, instantiates them, and provides limited runtime monitoring and logging.

5.1 A Comparison to the Serverless Framework

To get a better feel of UniK's capabilities we compared it with the serverless [Serverless.com, 2016], function-as-a-service framework. Both aim to reduce the amount of system preparation and administration needed to be performed by software developers. To have a concrete test-case, we wrote a simple microservice that takes Amazon S3 credentials and returns a polynomial hash computed over a subset of an Amazon S3 bucket's contents. This operation was then deployed to the Amazon cloud.

We started with a serverless setup. The project consists of roughly two parts: the function's implementation and deployment metadata. We opted

[9]As noted by https://github.com/emc-advanced-dev/unik, date 2016-11-06.

for a simple Python program utilizing the Amazon Boto library as well as a single custom Python GF256 mathematical library. The metadata is used to map an incoming HTTP requests to the corresponding python function, e.g., to map the "GET /create_hash HTTP/1.1" operation to the deployed python function "create_hash". Thus the framework blends application-level logic with deployment issues. The working example can be found on github[10]. The code was deployed using the serverless command line interface. Additional serverless features that were utilized during development were command-line web-service invocation and the integrated logging facilities. The latter give information about function runtime, which is important as Amazon Lambda functions are billed based on call count and length of function execution. All those features worked out-of-the-box, not too surprising given that we relied on Amazon-provided libraries to work within an Amazon-provided execution environment.

Then we turned to UniK's python support provided through a rumprun unikernel. We expected to deploy the same code base with slight modifications[11] into the Amazon cloud as an EC2 instance. In contrast to serverless, where a mapping from HTTP endpoint to python methods is utilized to create HTTP endpoints, our unikernel had to provide a web-server on its own. We opted for the python-included web server implementation and added a simple HTTP handler that then calls our original python function taken from our serverless code. This has the benefit, that we were able to easily test our microservice locally by just starting the web server. We were able to build and deploy a unikernel image to the Amazon cloud but alas the microservice did not work as expected and threw an exception. UniK handles virtual machines, not functions, so no logging facilities provided any help. We manually improved the functions logging and were thus finally able to deduce that the Boto library internally uses python's multiprocessing package. As mentioned before, unikernels do not generally support multiple execution flows, on which multiprocessing depends—thus producing our microservice error. This is a deeply unsatisfying situation as, with python being a dynamically interpreted language, there is no fail-safe and easy way to detect such dependency problems prior to deployment. This makes selection of libraries for unikernel projects risky and, in our opinion, favours statically compiled languages for unikernel.

[10]repository available at https://github.com/andreashappe/serverless-python-aws.

[11]repository available at https://github.com/andreashappe/unik-python-aws.

5.2 Is This Even Within UniK's Scope?

This is not directly UniK's fault, but happened due to the technology stack selection we unwittingly chose. UniK manages virtual machines, thus has no information about exported methods and their communication endpoints. It only provides logging on a whole VM base, i.e., provide a virtual console that captures and stores the virtual machine's "TTY" output, whereas serverless integrates on a higher application level, thus knows about deployed functions and can provide detailed information on this fine-granular level.

UniK creates a virtual machine that can be deployed on multiple backends provided by the user. This allows users to take back full control over the software and hardware stack. In contrast, serverless fully depends on the backend provider. In the Amazon Cloud case, no information about the used hardware or underlying software stack is available—deeply troubling some reliability engineers, especially when sensitive data should be handled by the deployed micro service.

6 THE VISION: THE BEST OF BOTH WORLDS?

Compared to existing UniK compilation frameworks, the serverless framework provides a better developer experience. Seeing UniK and serverless as competitors leads to a false dichotomy, we see both of them as part of a potential solution to the exploding complexity of software development and deployment.

6.1 Limit Unikernels and Integrate Them with their Environment

Serverless focuses on creating an environment for functions provided through an HTTP (web) interface and thus can provide specialized logging and monitoring information tailored for this use-case. UniK in contrast provides a compilation framework for generic operating systems—without further specialization it is not possible to gather detailed information for included services.

Unikernels provide a lean environment, focusing on a single application. We propose the same minimalism when it comes to protocols and their integration. Instead of providing generic containers, we propose that a single unikernel should be tailored to provide the endpoint for a single web request. To achieve this, additional external and internal components will need to be devised. Externally a smart communication endpoint for HTTP traffic must be introduced.

This endpoint will take incoming HTTP/HTTPS requests, extract the encapsulated request, match it against provided unikernels and then forward a sanitized operation towards the encapsulated operation within the unikernel. It will be the starting point for additional features such as providing unikernels on demand, i.e., starting a new unikernel if an operation request was received. As dispatched operations are HTTP requests, logging can be specialized towards the needs of this protocol. As seen with our UniK experiments, currently each unikernel has to implement a whole web server stack on its own. We propose a small library for usage within the unikernel that would accept an incoming web request from the external HTTP server and calls the corresponding function within the unikernel. This library would also include the logging interface needed for advanced data gathering. As each unikernel function will always be called on behalf of an incoming request we assume that we can gather logging information on a per-request basis.

The incoming (external) web-server is the perfect place to implement SSL/TLS termination. This allows for hot-plug of existing unikernels as the TLS connection will be created between the client and the HTTPS listener (and not between the client and the unikernel). We have chosen the Go programming language for implementation as this is also the programming language used by the UniK project and we target close interactions with their development community.

As communication between the external web server and the unikernel is based on a technology agnostic protocol (HTTP), we can couple multiple unikernel implementations with a single external web server. Due to this, we name the external web server a “multi-unikernel web server”.

6.2 Tooling/Further Hardening of Unikernels

Our tests with a python-based unikernel have shown that tooling is of the highest importance when it comes to successful unikernel adoption. While dynamic programming languages commonly yield higher developer efficiently, due to our problems with Python, i.e., not being able to detect incompatible libraries during development time, we lean towards usage of statically compiled programming languages. This allows us to perform all dependency and compatibility checks during compile time. We mandate that the framework must provide and enforce a sensible selection of defensive libraries for input data validation and output sanitization.

When it comes to programming language selection, we assume that either the programming language is memory-safe itself or the tooling provides for vast amounts of static source code analysis during compile- and run-time to mitigate potential vulnerabilities. In addition, a sensible selection of libraries should be utilized to reduce negative security implications (i.e., memory management). If source code is statically linked (which we imply), steps should be taken to improve the resilience of code against security attacks. There is a large repository of address randomization [Shacham et al., 2004], and code-execution protection [Childs Jr et al., 1984], techniques which should be employed automatically.

Note, that additional security libraries might introduce additional vulnerabilities. We propose that automatic minimization offered by our unikernel-system will offset this negative security impact. Static source code analysis and defensive measures like address-randomization do not add additional code to the compiled binary and thus do not increase the attack surface.

6.3 Marry the Serverless Developer Experience with Unikernels

So far, we talked about infrastructure mechanisms that integrate unikernel compilation, deployment and monitoring. In addition we forsee potential improvements with regard to developer usability by integrating our web server/UniK stack backend with serverless’ developer frontend.

Initially, we proposed a multi-unikernel web server that allows for hot-plugging web-based unikernels. The second step focused on improving the compilation step of a single unikernel through usage of defensive security libraries. This final step will integrate with the serverless framework. A serverless project consists of multiple functions and attached metadata, i.e., under which HTTP address which operation should be available. We propose to take this information and use it to create a single web-based unikernel per exported function. Information needed for deployment within out multi-unikernel web server will be extracted from serverless’ meta-data. Our logging subsystem will also be integrated with serverless’ logging front-end.

7 CONCLUSION AND FUTURE WORK

We have shown how the use of unikernel based systems can be used to reduce complexity, and thus improve security. We have looked at real world empir-

ical security findings as a foundation for this work, and have been able to highlight how unikernels can be used to mitigate security risks. We discussed how unikernels will form an integral part of the overall software architecture and examined patterns for that architecture which can be used to aid software development. We therefore identified the requirement for a unikernel-enabling surrounding framework to provide better support for the unikernel approach, and to that end we discussed how this might be achieved, and detailed a high- level architecture. We analyzed the current state of unikernel compilation frameworks, identified their shortcomings and offered sensible future paths which could support powerful and scalable multi-unikernel systems.

We are currently analysing different static and dynamic hardening mechanisms that can be added during unikernel compilation. The result of this analysis will be a paper detailing the different potential mechanisms which could be used and their impact on runtime performance, unikernel image size as well as runtime memory consumption. Early indications suggest this work is likely to be of benefit. In parallel we are investigating the use of a dedicated unikernel web server that allows for on-demand launching of unikernels for scale-out. To achieve this in a developer friendly manner, additional in-unikernel libraries will have to be devised. This will likely take some effort, but we believe this will greatly enhance the prospect that a more secure approach could be achieved.

We have also carried out some preliminary work on the use of unikernel systems to assist in dealing with some of the worrying security weaknesses in IoT technology, and we also continue with this work. Based on the work in this, and three previous papers—[Duncan et al., 2016a], [Bratterud et al., 2017], and [Duncan et al., 2016b]—we are convinced that properly developed unikernel-based solutions can provide a vital weapon in the armoury of enterprises for ensuring improved levels of security can be both achieved and maintained.

ACKNOWLEDGEMENTS

This work was in part funded by the European Commission through grant agreement no 644962 (PRISMACLOUD).

REFERENCES

37signals. Make Opinionated Software.

Abadi, D. J. (2012). Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer (Long. Beach. Calif).*, (2):37–42.

Anati, I., Gueron, S., Johnson, S., and Scarlata, V. (2013). Innovative technology for CPU based attestation and sealing. In *Proc. 2nd Int. Work. Hardw. Archit. Support Secur. Priv.*, volume 13.

Arce, I. (2004). The shellcode generation. *IEEE Secur. Priv.*, 2(5):72–76.

Bass, L., Weber, I., and Zhu, L. (2015). *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

Bhatkar, S., DuVarney, D. C., and Sekar, R. (2005). Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Usenix Secur.*

Blankstein, A. and Freedman, M. J. (2014). Automating isolation and least privilege in web services. In *Secur. Priv. (SP), 2014 IEEE Symp.*, pages 133–148. IEEE.

Bonér, J., Farley, D., Kuhn, R., and Thompson, M. (2014). The reactive manifesto.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998). Making the future safe for the past: Adding genericity to the Java programming language. *Acm sigplan Not.*, 33(10):183–200.

Bratterud, A., Happe, A., and Duncan, B. (2017). Enhancing Cloud Security and Privacy: The Unikernel Solution. In *Cloud Comput. 2017 Eighth Int. Conf. Cloud Comput. GRIDs, Virtualization*, pages 1–8.

Bratterud, A. and Haugerud, H. (2013). Maximizing hypervisor scalability using minimal virtual machines. In *Cloud Comput. Technol. Sci. (CloudCom), 2013 IEEE 5th Int. Conf.*, volume 1, pages 218–223. IEEE.

Bratterud, A., Walla, A.-A., Engelstad, P. E., Begnum, K., and Others (2015). IncludeOS: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th Int. Conf. Cloud Comput. Technol. Sci.*, pages 250–257. IEEE.

Bui, T. (2015). Analysis of docker security. *arXiv Prepr. arXiv1501.02967*.

Burbeck, S. (1992). Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc). *Smalltalk-80 v2*, 5.

Candea, G. and Fox, A. (2003). Crash-Only Software. In *HotOS*, volume 3, pages 67–72.

Childs Jr, R. H. E., Klebanoff, J. L., and Pollack, F. J. (1984). Microprocessor memory management and protection mechanism.

Climate, C. (2013). Rails' Remote Code Execution Vulnerability Explained.

Costan, V. and Devadas, S. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086.

Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. (1998). StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Usenix Secur.*, volume 98, pages 63–78.

Detlefs, D., Dosser, A., and Zorn, B. (1994). Memory allocation costs in large C and C++ programs. *Softw. Pract. Exp.*, 24(6):527–542.

Duncan, B., Bratterud, A., and Happe, A. (2016a). Enhancing Cloud Security and Privacy: Time for a New Approach? In *Intech 2016*, pages 1–6, Dublin.

Duncan, B., Happe, A., and Bratterud, A. (2016b). Enterprise IoT Security and Scalability: How Unikernels can Improve the Status Quo. In *9th IEEE/ACM Int. Conf. Util. Cloud Comput. (UCC 2016)*, pages 1–6, Shanghai, China.

Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., and Others (2014). The matter of heartbleed. In *Proc. 2014 Conf. Internet Meas. Conf.*, pages 475–488. ACM.

Fan, W. and Bifet, A. (2013). Mining big data: current status, and forecast to the future. *ACM sIGKDD Explor. Newsl.*, 14(2):1–5.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine.

Flatpak. The Future of Application Distribution.

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc.

Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317.

Jithin, R. and Chandran, P. (2014). Virtual machine isolation. In *Int. Conf. Secur. Comput. Networks Distrib. Syst.*, pages 91–102. Springer.

Kernighan, B. W., Ritchie, D. M., and Ejeklint, P. (1988). *The C programming language*, volume 2. prentice-Hall Englewood Cliffs.

Madhavapeddy, A., Leonard, T., Skjegstad, M., Gazagnaire, T., Sheets, D., Scott, D., Mortier, R., Chaudhry, A., Singh, B., Ludlam, J., and Others (2015). Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symp. Networked Syst. Des. Implement. (NSDI 15)*, pages 559–573.

Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. (2013). Unikernels: Library operating systems for the cloud. *ACM SIGPLAN Not.*, 48(4):461–472.

Madhavapeddy, A. and Scott, D. J. (2013). Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30.

Madnick, S. E. and Donovan, J. J. (1973). Application and analysis of the virtual machine approach to information system security and isolation. In *Proc. Work. virtual Comput. Syst.*, pages 210–224. ACM.

Martin, R. C. (1995). Principles of OOD. *Von butunclebob. com http//butunclebob. com/ArticleS. UncleBob. Princ. abgerufen*.

Marz, N. and Warren, J. (2015). *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co.

Meyerovich, L. A. and Rabkin, A. S. (2013). Empirical analysis of programming language adoption. *ACM SIGPLAN Not.*, 48(10):1–18.

OWASP. Attack Surface Analysis Cheat Sheet.

OWASP (2013). OWASP Top Ten Vulnerabilities 2013.

Pike, R. (2009). The Go Programming Language. *Talk given Google's Tech Talks*.

Rutkowska, J. (2013). Thoughts on Intel's upcoming Software Guard Extensions (Part 1). \url{http://theinvisiblethings.blogspot.co.at/2013/08/thoughts-on-intels-upcoming-software.html}.

Serverless.com (2016). Serverless Architectures.

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *Proc. 11th ACM Conf. Comput. Commun. Secur.*, pages 298–307. ACM.

Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Oper. Syst. Rev.*, volume 41, pages 275–287. ACM.

Stroustrup, B. (2015). *Die C++-Programmiersprache: aktuell zum C++ 11-Standard*. Carl Hanser Verlag GmbH Co KG.

Ubuntu. Snapcraf Overview.

Wadler, P. (1992). The essence of functional programming. In *Proc. 19th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, pages 1–14. ACM.

Yang, J. and Hawblitzel, C. (2010). Safe to the last instruction: automated verification of a type-safe operating system. In *ACM Sigplan Not.*, volume 45, pages 99–110. ACM.