Lateral Thinking for Trustworthy Apps

Hermann Härtig, Michael Roitzsch, Carsten Weinhold, Adam Lackorzyński
Technische Universität Dresden, Operating Systems Group, Dresden, Germany
{hermann.haertig,michael.roitzsch,carsten.weinhold,adam.lackorzynski}@tu-dresden.de

***Abstract*—The growing computerization of critical infrastructure as well as the pervasiveness of computing in everyday life has led to increased interest in secure application development. We observe a flurry of new security technologies like ARM TrustZone and Intel SGX, but a lack of a corresponding architectural vision. We are convinced that point solutions are not sufficient to address the overall challenge of secure system design. In this paper, we outline our take on a trusted component ecosystem of small individual building blocks with strong isolation. In our view, applications should no longer be designed as massive stacks of *vertically* layered frameworks, but instead as *horizontal* aggregates of mutually isolated components that collaborate across machine boundaries to provide a service. Lateral thinking is needed to make secure systems going forward.**

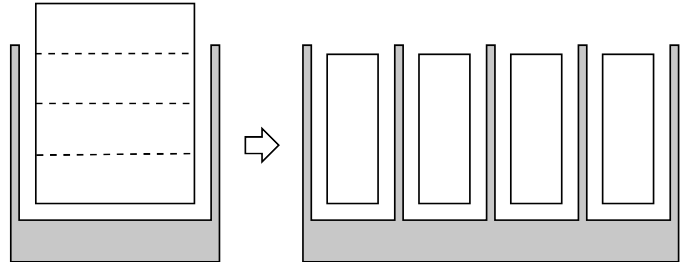

Fig. 1. Vertical and Horizontal Application Design

I. Introduction

The world around us is becoming digital. Critical infrastructure is increasingly controlled by computers and with the current IoT (Internet of Things) uptake, computerized fridges and light bulbs enter people's homes. Unfortunately, the current paradigms for application design require a lot of developer effort to protect all this software against malicious attackers, resulting in the countless security incidents we have seen over the last years. Lately, the DARPA Cyber Grand Challenge [1] has demonstrated the scale and urgency of the problem, awarding $2 million to the winner of a hacking tournament where security vulnerabilities are found and exploited automatically by machines. Attacks today increasingly focus on endpoint devices, because all the recent work on cryptographic communication protocols is paying off. Now we must act quickly to secure the endpoints.

To counteract our current systems' porous security, silicon vendors shipped novel hardware features that offer special protection for critical components to shield them from the large legacy codebases which are potentially compromised. Examples for such hardware techniques include ARM's TrustZone adding an extra security layer underneath kernel mode [2], Apple's Secure Enclave Processor running a variant of the L4 microkernel [3], and Intel's SGX [4], which adds complex isolation and authentication functionality to the hardware [5]. However, current application of these technologies is limited to tiny pieces of the system, often related to authentication, integrity checks, or cryptographic operations. We believe that unchaining the full potential of these new features requires a more radical change in how systems are designed and how applications are built.

Applications are currently constructed as monolithic blobs of *vertically* stacked frameworks. To satisfy the world's growing needs for software, developers rely on component reuse, which today is accomplished by adding existing libraries to applications. Such libraries provide a wealth of building blocks, from low-level system functionality like file and network access up to the parsing and rendering of HTML content. The resulting applications aggregate many different responsibilities in one binary and thus in one running process, making it fundamentally difficult to apply established security concepts like the Principle of Least Authority (POLA) effectively. An application that reads from the network and parses HTML can be subverted and its wide-ranging access privileges can compromise the system. Established belief that this can be fixed retroactively by additional filtering layers like intrusion detection and virus scanners is increasingly questioned, because these components themselves add to the attack surface [6].

Proposals for secure architectures [7], [8], [9] have long advocated for identifying security-critical pieces of applications and isolating them from the larger untrusted codebase. Unfortunately, this split-application approach requires manual code refactoring, so it is used reluctantly. Sandboxing as employed by web browsers [10] is one of the few mainstream successes inspired by this idea.

Instead of incremental refactoring, our vision is to turn the tables on application architecture. As illustrated in Figure 1, we want to carry the split-application approach to its natural conclusion. Instead of separated components being the exception only used manually for the most critical features, we postulate that separation should be the norm that is built right into the development workflow. Instead of vertically stacked libraries, we envision applications to be *horizontal* aggregates of communicating components, individually isolated from one another and mutually distrusting. Because the privileges of each component can be limited much tighter according to POLA, a subversion of one component can often be contained and does not infect other components. This stronger separation

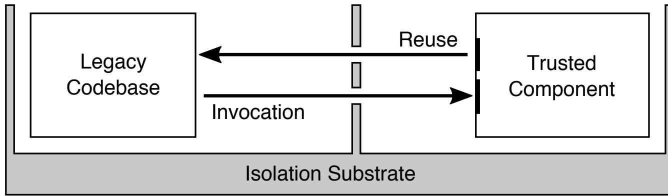

Fig. 2. Structural Template

of components can lead to fundamentally more secure systems. Later in the paper, we discuss such benefits using a common mail client as a first example. We identify components that should be isolated and their trust relationships.

By using trust anchors provided by the hardware, our envisioned architecture also extends across the network, allowing trusted component interaction in distributed systems. We discuss a smart meter example, where the grid operator needs to trust measurements by the client and the client wants to have secure and private interaction with the operator's server.

To explain our vision, we first provide in Section II an understanding of the state of the art in isolation technologies. We show that past as well as current developments are based on common principles, but with differing security properties. Also in Section II, we compare the different levels of hardware support and discuss, which hardware features are minimally required to implement individual isolation features. Section III explains our idea of a unified interface to these underlying isolation technologies. This interface should abstract from them in the same way POSIX abstracts different UNIX-style kernel implementations. We also present our paradigm for horizontal application development based on this interface and discuss example scenarios. To support developers in this new approach, we conclude the paper with a call to action for the research community: Section IV outlines the required improvements to security analysis and debugging tools, as well as programming language integration.

We believe our vision can help to realize an ecosystem of trusted components, which application developers and system integrators can reuse like libraries are reused today, but in a way that is fundamentally more secure and allows to develop the next generation of secure systems.

II. A Unified View on Hardware Isolation

Because isolation between trust domains is a major building block for secure systems, we give an overview of key representatives of existing isolation technologies. We start this section by describing a structural template to show that current hardware technologies are all instances of a common pattern. Later in the paper, we explain our envisioned application architecture using terminology we introduce with this structural template.

A. Structural Template

Figure 2 illustrates what we believe is a common pattern for isolation technologies. We identify four key elements:

Isolation Substrate: The isolation itself is anchored within an isolation substrate, which implements spatial as well as temporal isolation. Spatial isolation governs access to memory or other storage resources. Temporal isolation governs access to processing resources, which can range from simple starvation prevention to interference-free scheduling and covert channel mitigation. The isolation substrate allows for two or more distinct components that are subject to those isolation limits. Controlled communication channels implemented by the substrate allow those components to interact. Because it provides the mechanisms for securely isolating components, the substrate is assumed trusted. The protection is rooted in the hardware, but can include different amounts of software.

Legacy Codebase: Code following a traditional monolithic design combines different subsystems into one protection domain. Those subsystems may exhibit potentially asymmetric internal trust relationships, but because there is no isolation between these subsystems, these legacy codebases expose a large attack surface. Any security vulnerability within any subsystem can lead to a complete takeover of the entire legacy application. Therefore, legacy code is considered not trustworthy and assumed to be compromised. Legacy codebases can also contain entire operating systems. Unfortunately, we cannot expect to get rid of legacy code, because most of today's productive software falls into this category.

Trusted Component: A component can be considered trustworthy for various reasons. It may have been designed more rigorously, for example by properly separating responsibilities according to POLA. It may even be small enough to enable formal verification. However, trusted components do not necessarily have to be small and methodically constructed: In scenarios where code from different stakeholders is combined, trust may simply be a function of who controls which pieces. In cloud data centers, the cloud operator provides the untrusted runtime environment and the customer provides the trusted code. This customer code may be a large standard application, but the customer trusts it because he or she controls it. Trusted components rely directly on the isolation substrate to substantiate the trust placed in them. We say that the isolation substrate constitutes the component's Trusted Computing Base (TCB).

Communication: Legacy codebases and trusted components must be allowed to interact. When legacy code requires functionality from trusted components, some mechanism for service invocation is executed. On its interface to untrusted legacy code, the trusted component must make sure to not fall victim to confused deputy attacks [11]. The isolation substrate can provide a trust anchor to enable secure identification of communicating components. To fulfill its service, the trusted component may itself reuse functionality from the legacy codebase. To make sure this is safe, the trusted component must be considerate not to leak information and must carefully vet the reply. Cryptography may help to satisfy these requirements.

B. Existing Isolation Technologies

After having established the common structural template, we explain existing hardware isolation technologies in detail. These technologies are designed with different properties to defend against different attacker models, but they are all instances of the common template.

ARM TrustZone: ARM's TrustZone is a feature built into ARM CPUs that provides two separate execution contexts: The *secure world* context completely controls the software running in the *normal world* context [2]. The TrustZone hardware therefore acts as an isolation substrate, with untrusted legacy code being contained in the normal world and trusted components residing in the secure world. The normal world can invoke services from the secure world by *secure monitor calls*. Both worlds run their own operating system and can access hardware devices like interrupt controllers, caches, and memory controllers. TrustZone ensures that the hardware can distinguish which side is issuing an access by conveying an additional identifying bit with each request.

The TrustZone worlds are asymmetric, because the secure world exercises control over the normal world. Multiple trusted components may share the secure world, but then they rely on secondary isolation by the secure world operating system. TrustZone itself offers only a single secure world. The normal world can host exactly one legacy codebase, because TrustZone itself does not support multiplexing [12]. However, TrustZone can be combined with virtualization techniques to host multiple normal world operating systems [13]. The hypervisor software is then part of the isolation substrate.

Today, TrustZone is commonly used to implement security functionality in smartphones and tablets. The normal world runs the legacy Android system, while the secure world provides security services like cryptography and digital rights management. The TrustZone isolation substrate protects keys in the secure world from access by the possibly compromised normal world. Under names like Qualcomm QSEE or Samsung Knox [14], smartphone vendors market their custom set of trusted components running within the secure world. Knox comes pre-installed on Samsung's Galaxy product line of phones and tablets. It offers features such as integrity measurement of the running Android Linux kernel to detect any abnormal behavior and thus protects against kernel intruders.

Apple Secure Enclave Processor: Apple's iOS devices and most recent laptops employ a security coprocessor called the Secure Enclave Processor (SEP). The SEP is separated from the main application CPU, accesses DRAM with inline encryption and runs an L4-style microkernel [3]. It hosts cryptographic services and handles fingerprint authentication. Sensitive keys and biometric material is protected from the potentially compromised application CPU. The hardware separation and the communication bus between SEP and CPU thus form the isolation substrate, with the SEP hosting trusted components that serve the legacy codebase on the application processor. By using a dedicated processor, this construction offers strong isolation with reduced side channel opportunities compared to shared-hardware solutions. But similar to TrustZone, SEP is inflexible and offers only two separated execution environments.

Similar properties can be achieved in data center operation by using tamper-resistant processors in the form of Hardware Security Modules (HSM). The SEP is essentially an on-device HSM.

Trusted Platform Module: In 2003, the PC industry started shipping laptops with a security chip called the Trusted Platform Module (TPM) [15]. The purpose of a TPM is threefold: First, it stores cryptographic keys for encryption and digital signatures in hardware, where they cannot be leaked or stolen by software running on the main processor. Second, the TPM provides means to restrict access to these keys to specific software stacks, namely those whose overall code base match a predetermined cryptographic checksum. And third, it can also digitally sign this checksum in order to attest to a remote party, which software stack has been booted.

According to our structural template, the services implemented by the TPM constitute a trusted component with a hardware trust anchor. Isolation is provided by physical separation of the TPM chip from the main CPU. All the code on the main CPU is considered untrusted and blocked from accessing the cryptographic keys within the TPM. Because of their fixed-function nature, TPMs are in limited practical use. One prominent example is Microsoft's BitLocker feature, where the TPM releases the full-disk-encryption key of the system volume only to a correct version of Windows that has not been tampered with.

To reduce the amount of software that is part of the attested software stack, AMD and Intel introduced processor and chipset support to securely launch code even after the system has been booted. The TPM can attest this code to a remote party without the need to include the entire boot chain, which includes large codebases like BIOS, boot loader, and legacy kernel. This feature is called *late launch* and can be activated by executing a special CPU instruction. This instruction causes all currently running software including the kernel to be stopped, before a small piece of code is given full control over the machine. Upon executing the late-launch instruction, the CPU and chipset report the cryptographic hash of this piece of code to the TPM, which can later attest the identity of the newly bootstrapped code, as well as provide exclusive access cryptographic keys based on this identity. The Flicker project [16] has demonstrated that late launch can be used as an isolation mechanism to execute trusted components from within legacy code. Flicker even allows multiple trusted components that are mutually isolated by way of the TPM assigning them different cryptographic identities, but they cannot run concurrently.

Intel SGX: Intel's Software Guard Extensions (SGX) can be regarded as a more refined implementation of the late-launch approach, where independent trusted components can run concurrently in their own fully isolated *enclaves* [4]. The operating system can schedule enclaves similarly to how it

assigns CPU time to threads in ordinary processes, but only the code running inside an enclave can see and manipulate the memory that has been allocated to it. SGX hardware in the CPU transparently encrypts and decrypts the enclave memory, which is backed by DRAM. Because SGX also includes TPM-like cryptographic functions such as attestation, a trusted component running inside an enclave relies on an isolation substrate that comprises just the Intel CPU. Similar to Intel SGX, AMD ships a technology called SEV [17], which transparently encrypts memory used by virtual machines and also implements a hardware trust anchor.

SGX is recently gaining a lot of research interest [18], [19], [20], [21] because it also enables another interesting use case: When running software on rented servers within a data center, SGX allows to run the code without the server operating system or data center staff having any visibility into the execution state. The data center customer needs to trust only the Intel CPU, but not the operating system nor any other software outside of his enclave. The trusted component in such a setup is running a potentially large amount of code, which is trusted by the customer, simply because this code is fully under the customer's control. Reuse of services offered by the legacy operating system outside the enclave is possible, but needs to be done with care to not fall victim to corrupted responses.

Operating-System-Based Separation: The isolation technologies we presented so far were largely hardware-based, ranging from baked-in trusted functionality like the TPM to fully programmable enclave environments like TrustZone, SGX, or the SEP. But another line of isolation implementations should not be forgotten: the traditional operating system and hypervisor constructions, which rely on a small hardware mechanism to enforce isolation, but where this mechanism is controlled by software to implement isolated protection domains.

A lightweight software-based implementation are microkernels, which use the MMU to isolate processes from one another. Because they run spatially isolated by address spaces and temporally isolated by preemptive scheduling, these processes can host trusted components or legacy code alike [7]. To protect against malicious devices and drivers, memory accesses from devices can be filtered by an IOMMU. The MMU and IOMMU hardware together with the microkernel controlling them comprise the isolation substrate, which trusted components rely upon. However, microkernels can be formally verified [22], which significantly strengthens their trustworthiness.

MMU-based isolation can even run entire legacy operating systems using *paravirtualization* techniques [23]. This approach was used on ARM hardware to implement Simko3, the so-called *Merkel-Phone*, a smartphone that is based on the L4Re system [13]. The phone offers two Android systems side by side on the same phone, allowing the user to separate private and business use within one device. This separation is accomplished by running two virtual machines, each running its own instance of Android.

Other isolation mechanisms are hardware-assisted virtualization, which hypervisors use to separate unmodified legacy operating systems, or network-on-chip-based message isolation, which is used in research systems for heterogeneous manycores [24].

Pure Software Isolation: Components can also be isolated purely by constructing them using type-safe languages [25]. Code written in such a language is compiled to a binary that can no longer manipulate memory arbitrarily but only within the strict rules of the type system. This property can be leveraged to securely colocate components without any hardware isolation. The compiler of course must be trusted to enforce these rules and is therefore part of the TCB. Hybrid solutions like Google's Native Client [26] combine hardware isolation with compiler-enforced restrictions.

We do not consider software isolation solutions any further in this paper. Due to the need for a specific compiler, they do not efficiently handle legacy code. In addition, more sophisticated isolation features like secure boot or attestation require hardware support anyway.

C. Levels of Hardware-Support

As we have shown, many variants of isolation technologies are available. They make different design choices, where the implementation of the isolation substrate draws the line between trusted hardware and trusted software. To enable a deeper understanding of these options, we want to discuss the relevant trade-offs and clear up some misconceptions.

Is Hardware Better? In some circles, isolation primarily implemented by hardware is considered superior. We posit that this is a flawed assumption. There is no fundamental reason, why hardware mechanisms should be considered more trustworthy or more secure than software solutions. It is true that hardware may be better evaluated than the common application software. But formally verified microkernels like seL4 [22] offer at least as strong guarantees.

Using time partitioning and scheduler interference analysis [27], microkernels provide strong temporal isolation by mitigating covert channels. Hardware on the other hand is leaky [28], even high-profile security technologies such as SGX suffer from starvation issues and cache side-channels attacks [29].

And while software is often developed in the open, to underline its trustworthiness, hardware vendors typically operate as closed shops. Hardware is also notoriously hard to update, if bugs are found after deployment. Errata lists of the big CPU vendors [30] prove, that bugs and even security-critical issues do occur frequently.

What Is Hardware? Just because a feature is shipped by a hardware vendor also does not necessarily mean it is implemented in hardware. An SGX-enabled CPU for example is not only hardware. Portions of the cryptographic protocols and isolation mechanisms are implemented in substantial amounts of microcode, which is software that no one but Intel can

audit. In terms of complexity, an SGX-CPU therefore adds the equivalent of likely many thousands of lines of code to the TCB. With System Management Mode and the Management Engine, Intel processors enlarge the attack surface [31] with even more pieces of firmware, which is software dressed as hardware features.

The boundaries between hardware- and software-based isolation are further blurred by the fact that isolation technologies are partially interchangeable: Microsoft Surface tablets implement TPM functionality not using dedicated TPM security chips, but as software running within TrustZone [32]. The MIPS implementors offer an isolated environment called Omnishield, which offers features similar to TrustZone [33]. However, is not realized as a hardware extension, but as a virtual machine provided by a trusted hypervisor that is open-source, to enable independent auditing.

Is Virtualization Better? Another fallacy is the belief that virtualization offers stronger isolation than MMU-based address spaces. Both features can be used to implement complete spatial isolation, where one compartment cannot access memory of the other. The 'walls' between compartments are not 'stronger' when they are virtual machine walls. Address space walls are just as impenetrable by the software running behind it. Furthermore, because of complex hardware emulation, virtualization solutions actually expose a larger attack surface.

What virtualization does offer is the added benefit of being able to run an entire operating system without modifications inside an isolated compartment, while address spaces only support running a traditional process. Some kernels even implement both abstractions: virtual machines and address spaces. This allows running both legacy operating systems and isolated trusted components next to each other on top of the same system [34]. Whether such operating systems should then be called 'kernel'—a term reserved for systems with address spaces—or 'hypervisor'—a term used for systems with virtual machines—is merely an academic discussion.

D. Required Hardware Features

We observe that different hardware or software implementations are not trivially comparable regarding their isolation strengths. Rather, different solutions address different attacker models. The assumed capacity to execute attacks ranges from remotely exploiting software vulnerabilities to physical manipulation of the hardware. Therefore, we focus our discussion on the features that software requires from hardware to ensure proper isolation under increasingly stronger attacker models.

Basic Access Control: Efficient spatial isolation necessitates a way to control access to resources. The primary external resource used by the CPU is access to DRAM memory. Because DRAM access is critical for performance, control is enforced in hardware. Sometimes the isolation feature is complex like SGX, sometimes it is much simpler like an MMU. Because the MMU only enforces access control, it needs to be configured by software with a sound policy. Such software is therefore part of the isolation substrate. Two separate CPU privilege modes are required to separate software that can program the MMU from software that cannot. Today's MMUs are programmed using page tables, but software-loaded TLB implementations have also been built.

In typical computer architectures, peripheral devices are also capable of direct DRAM access in the form of DMA transfers. This property indirectly allows the driver software controlling those devices to manipulate arbitrary DRAM content, including page tables, even if this device driver was not privileged to program the MMU directly. To defend against malicious devices and malicious device drivers, IOMMUs control memory access by the device the same way MMUs control memory access by the CPU.

Physical Exposure of Data: Basic process isolation and a secure operating system like a microkernel offer strong defenses against remote attackers. However, physical attacks can target the memory bus to exfiltrate secrets or alter DRAM content. In such scenarios, off-chip wires are assumed to be accessible to attackers, but on-chip processing and memory such as caches can be shielded by way of tamper-resistant hardware design. Therefore, software has to be aware of physical memory properties and needs control over placement of data in memory. Some memory may be on-chip and can be used as is, whereas data going to off-chip memory over an exposed bus must be encrypted.

SGX for example operates unencrypted on CPU caches, but uses hardware encryption when evicting cache content to DRAM. The respective DRAM regions must be configured beforehand as protected enclave memory. A software implementation of such memory encryption is conceivable using on-chip scratchpad memory. Scratchpad content would be spilled to DRAM explicitly by software. Such transfers benefit from accelerated cryptographic operations, which are useful in many other scenarios as well. Therefore, with on-chip scratchpad memory and crypto hardware, SGX-style memory encryption could be implemented using for example ARM TrustZone or Apple's SEP.

Secure Launch: Knowing what software runs on a platform is crucial for establishing trust relationships. In usage scenarios, where hardware is assumed to be physically inaccessible to attackers, it suffices to install the software of your choice in a secure environment and setup a cryptographic secret like an SSH key for remote access. However, the problem gets harder, when physical intrusion is part of the attacker model. Then a *trust anchor* that cannot be altered is needed in the machine's boot process. The anchor must enforce a *launch policy* to oversee the software that is being run on the machine.

One prominent implementation of such an anchor is *secure booting:* An unchangeable piece of software gets to execute as the first step after power is turned on. Its job is to initiate the boot loader, but it will do so only after a successful check of a digital code signature. The machine will refuse to run improperly signed software. Once the signed boot loader is running, it in turn can check the signatures of the components it launches next in the boot chain. By successively validating

signatures, once the system is fully brought up, we know for sure that all running software has been correctly signed [35].

TPMs provide another, more involved implementation of a trust anchor: the Core Root of Trust for Measurement (CRTM). The CRTM also is an unchangeable piece of a TPM system and implements a launch policy called *authenticated booting:* At boot, it will calculate a hash sum of the boot loader code and store it in a TPM hardware register, before the boot loader is executed. Just as with secure booting, each step in the boot chain does the same: have the TPM calculate and store a hash of the next stage. But in contrast to secure booting, no signature checks are performed and no code is rejected. The TPM registers merely form a cryptographic boot log that can later be verified to reliably know what software is running. With authenticated booting, users have the freedom to run arbitrary code on their hardware, which is a desirable feature for open platforms.

We have seen that the difference between secure and authenticated booting is simply caused by different launch policies implemented by the trust anchor. The common hardware requirement for both is a component with physically unchangeable properties that initiates the launch chain. Such a secure launch must not necessarily occur at the initial boot of the machine. Invoking the launch mechanism later by special hardware instructions is the foundation of TPM late launch and SGX.

Attestation: Once software is initiated under a secure launch policy, outside parties may want to rely on the software identity of their communication partner to establish distributed trust relationships. Such outside parties may be remote clients connected by untrusted networks or other processors within the same system. With physical intrusion as an assumed attacker model, communication busses within a system must be considered untrusted networks as well, the difference merely is the length of the wires.

Cryptographically proving the presence of a software stack requires a tamper-resistant secret with restricted access. Software that can access the secret can prove knowledge of this secret to outside parties without revealing the secret itself. Without a secret, everything about the platform is known, so a complete software emulation is possible. Such an emulation can say one thing when asked what software it runs, but actually run something else. The same issue arises if a secret can be cloned or access to it can be proxied.

But if the secret is only available to trusted components that were started by a secure launch policy, the remote party can establish a trust chain: Proof of access to the secret could not be provided by an imposter as long as the integrity of the trust anchor is intact. Both TrustZone and SEP can restrict access to hardware keys, allowing software implementations of attestation. TPMs sign the recorded boot log with an unreadable key that is in turn signed by the TPM manufacturer. SGX provides attestation through a specially endowed *quoting enclave* that Intel provides.

Summary: We have identified four incremental hardware requirements to address different attacker models: Basic access control to memory is needed to isolate components from one another. Memory placement control and memory encryption mitigate physical manipulation of data. A trust anchor ensures properties of the launch process even under physical manipulation of code. A secret with restricted access allows attestation of code identity over untrusted channels.

As we have illustrated, the resulting isolation features are not exclusive to all-in-one hardware technologies like SGX. Given the minimum hardware requirements, many competing implementations using a mix of hardware and software are possible.

III. Trusted Component Ecosystem

We now present our vision how to integrate the isolation technologies in a holistic design methodology for secure and trustworthy systems.

A. Short Term Vision: Unified Thinking

Some of the isolation substrates rely on heavyweight hardware features and some employ trusted software to implement separation. We have shown that these technologies all follow a common structural pattern and only differ in design trade-offs regarding the addressed attacker model, development cost, production cost, or the openness for independent audit. Software components should be developed once against the common pattern and then should run on any isolation implementation. To enable component interchange, vendors, practitioners, and the research community should get together and work towards a common interface for isolation technologies. This interface should do for isolation mechanisms what POSIX did for the UNIX system call interface: allow application code to be independent of the underlying implementation. Such decoupling would allow system architects to reuse once developed components on different isolation primitives.

A unified interface also allows developers to hand-pick an isolation mechanism or combine multiple mechanisms based on the required attacker model. We think it is important that such choices are made deliberately and not based on fashionability of a new hardware feature, which may unnecessarily increase the attack surface by offering more features than needed.

Just an interface for isolation methods alone is not sufficient, because the isolated components need to be able to communicate to provide a service. The unified interface should be part of a larger programming framework, where developers can describe the required communication channels to other components [36]. Such a manifest enables the isolation substrate to establish just the needed channels and block all other communication, thereby promoting a POLA design mentality for the entire system. Furthermore, a map of communication relationships allows to reason about the required message protection if tampering is assumed.

B. Long Term Vision: Decomposition

Once a programming framework for component isolation is in place, the second stage of our vision is to reshape application design from a vertical stack of libraries to a horizontal aggregation of components. Applications should no longer use one address space to colocate subsystems that do not trust each other or that experience an asymmetric trust relationship. Distrusting subsystems should instead be separated into components so the substrate can enforce isolation.

Carving out trusted components from applications should no longer be limited to the low-hanging fruit like cryptographic operations or digital rights management. It should be a general mechanism that is applied wholesale to all subsystems of an application. No longer should you think only in terms of a large legacy codebase and small trusted components. An ideal application is entirely decomposed into a collection of individual components, where each component is only trusting the code and data it controls internally, but distrusts all other parts of the system. Cooperation is limited to narrow and well-defined communication interfaces. The isolation and communication control provided by the substrate enforces the mutual trust or distrust relationships between the components.

Decomposition ideas have been explored in the research community for some time as a split-application architecture [37] or as a solution to secure a large legacy codebase [38]. We propose it as a vision for all application development. This goal resembles Microsoft's Next-Generation Secure Computing Base (NGSCB) efforts to split the Windows kernel and system services into smaller isolated components. The NGSCB architecture was designed around TPMs as an anchor of trust, but never became a finished product.

Being such an ambitious approach, there is the chance for failure, so we looked into other communities for support. The concept of individual distrusting actors is theoretically covered by works on multilateral security [39]. As a software development paradigm, it has found its way into cloud computing in the form of microservices. These are individually isolated software agents, that collaborate with other microservices to implement large web applications. The motivation behind microservices is to improve scalability and fault tolerance, but they are an indication that relentless decomposition may be a viable programming model.

C. The Vision Applied: Examples

We present our approach discussing an email client as our first example. To demonstrate how the concept extends across the network in a scenario where different isolation mechanisms are combined, we continue with a smart meter example.

Email client: A mail application needs to establish a secure network connection to the mail server and understand complex protocols such as IMAP. It has to parse and display email messages, which may be formatted using HTML. Messages can contain images, videos, and other complex attachments, which the email client must be able to decode and present to the user. When composing a new email, users want to fill in recipient addresses stored in their address book. They edit text using traditional keyboard and mouse, by typing on a virtual touch-based keyboard on screen, or even by dictating the message. Any of these input methods can greatly benefit from highly personal data such as user dictionaries for spell checking, training datasets for voice recognition, or auto correction based on phrases and names previously used. Finally, a mail client needs to store messages in the file system, organize them in folders, search them, and display the user interface.

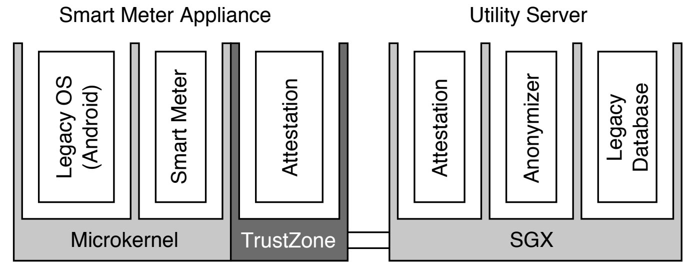

Fig. 3. Smart Meter Appliance and Utility Server

This wealth of functionality is typically implemented today by reusing existing code in the form of libraries. We propose to instead organize code reuse by isolated, but communicating components. The first steps of decomposing applications into components are already underway in today's commodity operating systems. The Mail app shipping with Apple's macOS uses the WebKit framework, which already employs a reusable out-of-process rendering component for HTML and CSS that cannot affect the mail application if compromised.

In a completely decomposed design, we would implement many more features in separate components. Input methods rely on contextual user data for recognition and autocorrection. Access to such data should be restricted to the input method code only. Code that handles handles data received from the network such as file format detection and rendering should be isolated, because it is exposed to attacks from the Internet. The networking part of the email client benefits from separation into a component handling application-level protocols such as IMAP or SMTP, and another component for transport-layer security (TLS) and login. If only the TLS component can access the device driver of the network card, the isolation substrate enforces mandatory encryption and integrity protection. Cryptographic keys and the user's account passwords are shielded from all other components.

Because the TLS component handles highly sensitive data, the application designer may decide it requires protection against a stronger attacker with physical access. The TLS component could be placed on a tamper-resistant substrate like the Apple SEP or inside an SGX enclave. Using a suitable trust anchor, it could verify the integrity of the component on whose behalf it is connecting to the email server.

Smart Meter Appliance: The TLS component in the above example already hints at the potential of this approach for extending trust relationships across machine boundaries. Figure 3

illustrates how the properties of multiple isolation substrates can be combined for a distributed trust application. A smart meter monitors electricity usage and sends real-time telemetry to the utility company. It receives feedback from the utility server on pricing changes and can selectively turn adaptive devices on or off. So on one hand, the smart meter handles highly privacy-sensitive data, which users expect the system to safeguard. On the other hand, the utility company relies on the delivered data for billing, so it expects the meter to be tamper-resistant.

To provide a complex user interface with touch input, our example smart meter runs a virtualized Android on top of a microkernel. The smart meter code itself is isolated as a separate component, such that Android vulnerabilities cannot harm the integrity and privacy of the meter readings.

When those readings are sent to the utility, the smart meter component wants to ensure the server will only use the data for billing purposes and afterwards stores only anonymized aggregates for long-term analysis. To ensure such properties of the server software, MMU-based isolation substrates are insufficient, because we must assume the utility could access the server and manipulate its software. As discussed above, attestation needs a trust anchor and hardware secret. The implementor may choose SGX because it provides those features and is widely available, even when the utility chooses to rent their server capacity from a cloud provider. SGX provides attestation by way of the quoting enclave. In this case the smart meter would verify the code identity of the data anonymizer component before sending it any readings. To encourage trust in its operation, the utility provider could open the source code of the anonymizer for third-party auditing. The smart meter would then check for the signature of the known-good anonymizer and refuse to talk to a manipulated instance that may violate user privacy.

The utility also needs to trust the meter readings, otherwise users could disconnect the actual meter and instead have a software emulation send fake data to the server. However, a weaker attacker model than complete physical compromise can be assumed, as most users have neither the equipment nor the expertise to pry open their device and manipulate the memory bus. The utility is more interested in keeping costs down, so they rely on embedded ARM processors and run software-implemented attestation within the TrustZone secure world. The attestation component is booted from read-only memory within the smart meter system-on-chip. This serves as a trust anchor to ensure confidence in the attestation code. A per-device AES key is fused into the chip by the manufacturer and is only accessible to the secure world, allowing the attestation component to prove its identity to the utility.

D. The Vision Realized: Reusable Components

The examples illustrate how multiple isolation substrates are combined to provide tailored protection for components and the communication channels between them. Applications are no longer monolithic blobs of co-located functionality, but aggregates of individually reusable components that can even form distributed confidence domains [40] across machine boundaries.

If component interactions are designed with POLA and multilateral security in mind, users can rely on *engineered privacy* instead of blind belief. The smart meter design outlined above empowers the user to know that customer data is handled anonymously. One still needs to trust the isolation substrate provided by SGX, but that is a much higher level of confidence than trusting wordy promises by the utility.

The smart meter example also demonstrates password-less authentication: The user is not entering a password into an Android app to view his billing information, but the appliance is authenticating itself using a secret hardware key. Because the user does not need to remember a credential, the system is resilient against phishing attacks, which are based on tricking the user into divulging credentials to the wrong party.

Network access of the Android subsystem can be filtered by an isolated gateway component. If this gateway has exclusive access to the network hardware, it can reliably enforce domain whitelists and bandwidth policies to prevent the smart meter appliance from participating in distributed denial-of-service attacks — an unfortunate reality with today's IoT devices.

These use cases and scenarios are not limited to the concrete examples of mail client or smart meter. Instead, they will likely appear in many applications and should be provided as reusable components. Once a unified interface for composition across substrates is in place, these components must only be implemented once and can be aggregated by configuring communication relationships between them.

Looking back at the structural template from Figure 2, we can identify the communication interfaces of trusted component to the outside as potential weak points: When the trusted component is invoked and exercises its authority on behalf of another, it may fall victim to a confused deputy attack. When the trusted component uses services from another part of the system, it may leak data to it or be confused by the reply. These problems can be addressed and the solutions should be part of the trusted component toolbox.

Confused Deputy: The confused deputy problem occurs when the same trusted component instance may serve multiple clients and thereby handle multiple trust domains within itself. If the code of the component is not carefully written, it may inadvertently confuse one client for another and thereby exercise the wrong session context.

Capabilities bundle communication right and context identification in one entity and are therefore an important programming tool to prevent confused deputy issues. The research community even discusses architectures with hardware capabilities to enable even more fine-grained disaggregation of authority. The CHERI [41] capability system is implemented as a modified MIPS CPU, using guarded pointers as capabilities.

Trusted Reuse: The other dangerous interface of a trusted component is reuse of untrusted functionality. Such an interface must be protected by a trusted wrapper [42], which are interface-specific and best explained with an example: Many

components have a need to store data, but the file system stack, including the storage device layer, is one of the most complex OS services. Their code bases comprise in the order of tens of thousands of lines of code and are therefore likely to contain exploitable weaknesses. Thus, trusted components should not rely on file system code to maintain data integrity or confidentiality.

The Virtual Private File System (VPFS) [43], [44] is a trusted wrapper allowing secure reuse of a legacy file system stack. The legacy stack takes care of actually storing file contents and managing the storage medium, but it never handles plaintext data. Instead, the VPFS wrapper guarantees confidentiality and integrity of all file system data and metadata by means of encryption and message authentication codes.

Secure Path to the User: Another vulnerable interface we identify is the one to the user. When multiple components in the system can interact with the user, it can be important to securely indicate which one is currently active [45]. Otherwise, it is the user who falls victim to a confused deputy attack by the system, which can be used for phishing. We have shown in the smart meter example, how systems can be engineered to be resilient to phishing. But if security relies on the user identifying the interaction partner, experience from web browsers has shown that this is a hard problem to solve [46]. Very obvious indication of a secure mode, like a simple traffic-light display may be advisable.

E. Potential Roadblocks

Even if decomposition puts up more walls and thereby limits the impact of misbehaving components, writing secure code within one component is still hard. With buffer overflows being mitigated by compiler and language protection, we expect confused deputy problems to be the new vulnerability du jour. Teaching developers to avoid them will require careful education.

The decomposition mentality itself can also complicate software development, because it adds to the mental burden for software engineers. Platform security by extensive communication control causes things to not work that would have worked without it. Apple is increasingly seeing developers leaving its Mac App Store, because the mandatory application sandboxing is complicating development too much.

However, we believe the benefits clearly outweigh these difficulties. The alternative is to continue with our post mortem patching culture, hoping to not fall too far behind the increasingly sophisticated attack methods. If we as a research community could get behind this effort, we could certainly further the adoption of this vision.

IV. Call to Action

We believe the recent development of new hardware isolation technologies as well as proven system software approaches like microkernels can be leveraged to establish a unified architecture based on horizontal interaction of trusted components. Common interfaces enable an open infrastructure for component reuse.

Developers need support for application decomposition by better programming language integration of address space separation. Existing approaches [47], [48] should be extended. Better tooling is needed to analyze security properties when applications consist of many independently communicating services. Especially, tools to uncover confused deputy problems are crucial, as these may become the primary attack vector against those systems.

Our proposed architecture can provide substantial instead of incremental improvements to system security. We imagine applications created by developers picking from a toolbox of horizontally aggregated components instead of vertically layering libraries. We invite system designers, application implementors, and the research community to join us and realize our vision of lateral thinking for a trustworthy application design.

Acknowledgment

This work has been been partially supported by the European Union through FP6 and FP7 grants and the ESF-funded projects IMData and microHPC, as well as by the German Research Foundation (DFG) projects SFB 358 and SFB 912, VPFS2, FFMK, Asteroid, and the Cluster of Excellence Center for Advancing Electronics Dresden (cfAED).

References

[1] M. Walker, “Defense advanced research projects agency: Cyber grand challenge,” http://www.darpa.mil/program/cyber-grand-challenge, 2016.

[2] *ARM Architecture Reference Manual*, Armv7-a and armv7-r ed., ARM Ltd., 2014.

[3] *iOS Security Guide*, iOS 9.3 ed., Apple, Inc., May 2016. [Online]. Available: https://www.apple.com/business/docs/iOS_Security_Guide.pdf

[4] *Intel® Software Guard Extensions – Developer Guide*, Intel Corporation, 2016. [Online]. Available: https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf

[5] V. Costan and S. Devadas, “Intel SGX explained,” Cryptology ePrint Archive, Computer Science and Artificial Intelligence Laboratory, MIT, Tech. Rep., 2016.

[6] R. O’Callahan, “Disable your antivirus software (except microsoft’s),” http://robert.ocallahan.org/2017/01/disable-your-antivirus-software-except.html, January 2017.

[7] H. Härtig, “Security architectures revisited,” in *Proceedings of the 10th ACM SIGOPS European Workshop*. ACM, September 2002, pp. 16–23.

[8] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth, “Reducing TCB complexity for security-sensitive applications: Three case studies,” in *Proceedings of the 1st ACM European Conference on Computer Systems*, ser. EuroSys. ACM, April 2006, pp. 161–174.

[9] N. Feske, *GENODE Operating System Framework 16.05*. Genode Labs, May 2016.

[10] C. Reis and S. D. Gribble, “Isolating web programs in modern browser architectures,” in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys. ACM, April 2009, pp. 219–232.

[11] N. Hardy, “The confused deputy (or why capabilities might have been invented),” *SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, October 1988.

[12] T. Frenzel, A. Lackorzyński, A. Warg, and H. Härtig, “ARM TrustZone as a virtualization technique in embedded systems,” in *Proceedings of 12th Real-Time Linux Workshop*, ser. RTLWS. OSADL, October 2010.

[13] M. Lange, S. Liebergeld, A. Lackorzyński, A. Warg, and M. Peter, “L4Android: A generic operating system framework for secure smartphones,” in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM. ACM, October 2011, pp. 39–50.

[14] “Samsung knox technology,” Samsung, https://www.samsungknox.com/en/knox-technology.

[15] *TPM Main Specification*, Version 1.2, revision 116 ed., Trusted Computing Group, March 2011. [Online]. Available: https://trustedcomputinggroup.org/tpm-main-specification/

[16] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the 3rd ACM European Conference on Computer Systems*, ser. EuroSys. ACM, March 2008, pp. 315–328.

[17] D. Kaplan, J. Powell, and T. Woller, *AMD Memory Encryption*, AMD, April 2016. [Online]. Available: http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[18] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI. USENIX, http://dl.acm.org/citation.cfm?id=2685048.2685070 2014, pp. 267–283.

[19] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using intel SGX," in *Proceedings of the 17th International Middleware Conference*. ACM, December 2016, pp. 14:1–14:13.

[20] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI. USENIX, November 2016, pp. 533–549.

[21] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure linux containers with intel SGX," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI. USENIX, November 2016, pp. 689–703.

[22] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, ser. SOSP. ACM, October 2009, pp. 207–220.

[23] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-kernel-based systems," in *Proceedings of the 16th ACM Symposium on Operating System Principles*, ser. SOSP. ACM, October 1997, pp. 66–77.

[24] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. Fettweis, "M3: A hardware/operating-system co-design to tame heterogeneous manycores," in *Proceedings of the 24st International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS. ACM, April 2016, pp. 189–203.

[25] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 37–49, April 2007.

[26] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, January 2010.

[27] M. Völp, B. Engel, C.-J. Hamann, and H. Härtig, "On confidentiality-preserving real-time locking protocols," in *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS. IEEE, April 2013, pp. 153–162.

[28] Q. Ge, Y. Yarom, and G. Heiser, "Contemporary processors are leaky – and there's nothing you can do about it," in *The Computing Research Repository*. arXiv, December 2016, version 3.

[29] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA. Springer, July 2017.

[30] *6th Generation Intel® Processor Family – Specification Update*, 332689-009EN ed., Intel Corporation, January 2017.

[31] S. Embleton, S. Sparks, and C. Zou, "SMM rootkits: A new breed of OS independent malware," in *Proceedings of the 4th International Conference on Security and Privacy for Communication Networks*, ser. SecureComm. ACM, September 2008.

[32] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, "fTPM: A software-only implementation of a TPM chip," in *Proceedings of the 25th USENIX Security Symposium*. USENIX, 2016, pp. 841–856.

[33] *Omnishield: An Overview & Requirements*, Imagination Technologies Limited, November 2016. [Online]. Available: https://www.imgtec.com/?do-download=omnishield-white-paper-an-overview-requirements

[34] A. Lackorzynski and A. Warg, "Taming subsystems: Capabilities as universal resource access control in L4," in *Proceedings of the 2nd Workshop on Isolation and Integration in Embedded Systems*, ser. IEES. ACM, March 2009, pp. 25–30.

[35] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 265–310, November 1992.

[36] H. Härtig and L. Reuther, "Encapsulating mobile objects," in *Proceedings of the 17th International Conference on Distributed Computing Systems*, ser. ICDCS. IEEE, May 1997, pp. 355–362.

[37] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzyński, F. Mehnert, and M. Peter, "The Nizza secure-system architecture," in *Proceedings of the 2005 International Conference on Collaborative Computing*, ser. CollaborateCom. IEEE, December 2005, pp. 10–19.

[38] D. G. Murray, G. Milos, and S. Hand, "Improving Xen security through disaggregation," in *Proceedings of the 4th International Conference on Virtual Execution Environments*, ser. VEE. ACM, March 2008, pp. 151–160.

[39] K. Rannenberg, "Multilateral security: A concept and examples for balanced security," in *Proceedings of the 2000 Workshop on New Security Paradigms*, ser. NSPW. ACM, September 2000, pp. 151–162.

[40] W. E. Kühnhauser, "Confidence domains for distributed systems," in *Proceedings of the 6th Canadian Computer Security Symposium*, 1994, pp. 177–200.

[41] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceeding of the 41st International Symposium on Computer Architecuture*, ser. ISCA. IEEE, 2014, pp. 457–468.

[42] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro, "Reducing TCB size by using untrusted components: Small kernels versus virtual-machine monitors," in *Proceedings of the 11th ACM SIGOPS European Workshop*. ACM, September 2004.

[43] C. Weinhold and H. Härtig, "VPFS: Building a virtual private file system with a small trusted computing base," in *Proceedings of the 3rd ACM European Conference on Computer Systems*, ser. EuroSys. ACM, May 2008, pp. 81–93.

[44] ——, "jVPFS: Adding robustness to a secure stacked file system with untrusted local storage components," in *Proceedings of the 2011 USENIX Annual Technical Conference*, ser. ATC. USENIX, June 2011, pp. 369–382.

[45] N. Feske and C. Helmuth, "A nitpicker's guide to a minimal-complexity secure GUI," in *Proceedings of the 21st Annual Computer Security Applications Conference*, ser. ACSAC. IEEE, December 2005, pp. 85–94.

[46] E. Gabrilovich and A. Gontmakher, "The homograph attack," *Communications of the ACM*, vol. 45, no. 2, p. 128, February 2002.

[47] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *Proceedings of the 13th USENIX Security Symposium*. USENIX, August 2004, pp. 57–72.

[48] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," in *Proceedings of 21st ACM Symposium on Operating Systems Principles*, ser. SOSP. ACM, 2007, pp. 31–44.