Sustainable Security & Safety: Challenges and Opportunities

Andrew Paverd
Aalto University, Finland
andrew.paverd@ieee.org

Marcus Völp
University of Luxembourg
marcus.voelp@uni.lu

Ferdinand Brasser
TU Darmstadt, Germany
ferdinand.brasser@trust.tu-darmstadt.de

Matthias Schunter
Intel Labs, Germany
matthias.schunter@intel.com

N. Asokan
Aalto University, Finland
asokan@acm.org

Ahmad-Reza Sadeghi
TU Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

Paulo Esteves-Veríssimo
University of Luxembourg
paulo.verissimo@uni.lu

Andreas Steininger
TU Wien, Austria
steininger@ecs.tuwien.ac.at

Thorsten Holz
Ruhr-University Bochum, Germany
thorsten.holz@rub.de

ABSTRACT

A significant proportion of today's information and communication technology (ICT) systems are entrusted with high value assets, and our modern society has become increasingly dependent on these systems operating safely and securely over their anticipated lifetimes. However, we observe a mismatch between the lifetimes expected from ICT-supported systems (such as autonomous cars) and the duration for which these systems are able to remain safe and secure, given the spectrum of threats they face. Whereas most systems today are constructed within the constraints of foreseeable technology advancements, we argue that long term, i.e., *sustainable security & safety*, requires anticipating the unforeseeable and preparing systems for threats not known today. In this paper, we set out our vision for sustainable security & safety. We summarize the main challenges in realizing this desideratum in real-world systems, and we identify several design principles that could address these challenges and serve as building blocks for achieving this vision.

1 INTRODUCTION

With the exception of a handful of systems, such as the two Voyager spacecraft control systems and the computers in the US intercontinental ballistic missile silos,[1] information and communication technology systems (ICT) rarely reach a commercially viable lifetime that ranges into the 25+ years we have come to expect from long-lived systems like cars[2], as shown in Figure 1. Worse, rarely any networked ICT system stays secure over such a lifetime, even if actively maintained.

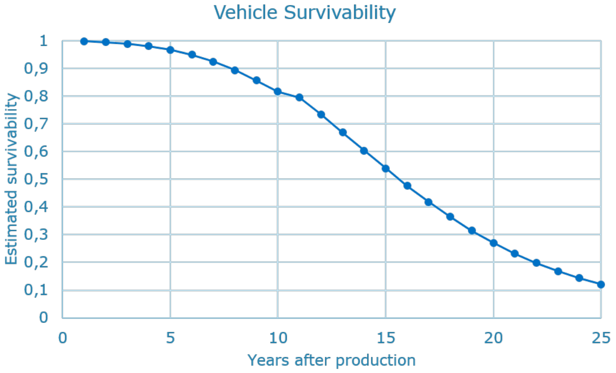

Figure 1: Survival probability of passenger cars by age [2]

Despite this potential risk, current ICT subsystems are already being integrated into systems with significantly longer design lifetimes. For example, electronic control units in modern (self-driving) cars, the networked control systems in critical infrastructure (e.g., cyber-physical systems in power plants and water treatment facilities), and computer controlled building facilities (a.k.a. smart buildings) are all examples of this mismatch in design lifetimes. This is also applicable beyond cyber-physical systems: for example, genomic data is privacy-sensitive for at least the lifetime of the person (if not longer), and yet it is being protected by cryptographic algorithms with a significantly shorter expected lifespan [14]. For the currently envisioned limited lifetime, the (functional) safety community has developed techniques to prevent harm despite accidental faults [23]. However, for the ICT systems in the above scenarios we aim to preserve security and safetime for the design lifetime, and often beyond. This means that they need to preserve the value and trust in the assets entrusted to them, their safety properties (i.e., resilience to accidental faults) *and* their security properties (i.e., resilience to malicious faults, such as targeted and persistent attacks). Even when compared to existing approaches to achieve resilience (such as the safety approaches mentioned above), these lifetimes translate into ultra-long periods of secure and safe operation.

To address this impending challenge, we introduce a new paradigm, which we call *sustainable security & safety* (S3). The central goal is that a system should be able to maintain its security and safety, but desirably also its functionality for at least its design lifetime. This is particularly relevant for systems that have significantly longer design lifetimes than their ICT subsystems (e.g., modern cars, airplanes, and other cyber physical systems). In its broadest sense, S3 encompasses both technical and non-technical aspects, and some of the arising challenges span both aspects.

[1] http://www.dailymail.co.uk/news/article-2614323/Americas-feared-nuclear-missile-facilities-controlled-computers-1960s-floppy-disks.html
[2] https://www.aarp.org/money/budgeting-saving/info-05-2009/cars_that_last.html

From a technical perspective, S3 brings together aspects of two well-studied technical fields: *security* and *dependability*. Security research typically begins by anticipating a certain class of adversary, characterized by a threat model based on well-known pre-existing threats at the time the system is designed. From this model, defense mechanisms are then derived for protecting the system against the anticipated adversaries or for recovering from these known attacks. Dependability, on the other hand, begins with the premise that some components of a system will fail, and investigates how to tolerate faults originating from these known subsystem failures. Again, the type, likelihood, and consequences of faults are assumed to be known and captured in the fault and fault-propagation models.

The term *security-dependability gap* [30] is a prototypical example of why established solutions from the dependability field cannot be directly used to address security hazards, and vice-versa. The problem of sustainable security has been recognized by other established researchers, as well, stating that safety and security will be more interweaved, evoking new research directions: For instance, to keep software-driven systems, like modern cars, secure over long periods of time they need to be supplied with software patches for decades, which is currently an unsolved problem [3].

These problems are not only challenging from technological perspectives, but also require the reorganization and adaption of non-technical aspects, like safety regulations, standardization bodies, and law making and enforcement [3]. Indeed many standardization and certification processes lack security consideration, a fact that was recognized by the European Commission, leading to an investigation of open problems and opportunities in today's area of Internet of Things [28, 29].

The security-dependability gap arises from the way risks are assessed in safety-critical systems: safety certifications assess risks as a combination of stochastic events, whereas security risks arise as a consequence of intentional malicious adversaries, and thus cannot be accurately characterized in the same way. What is a residual uncertainty in the former, becomes a strong likelihood in the latter.

Therefore, there are two defining characteristics of S3:

- Firstly, it aims to *bridge the safety-security gap* by considering the complementarity of these two fields as well as the interplay between them, and addressing the above-mentioned problems under a common body of knowledge, seeking to prevent, detect, remove and/or tolerate both accidental faults and vulnerabilities, and malicious attacks and intrusions.
- Secondly, it aims to *protect systems beyond the foreseeable horizon* of technological (and non-technological) advances and therefore cannot be based solely on the characterizations of adversaries and faults as we know them today. Instead we begin from the premise that a long-lived system will face changes, attacks, and accidental faults that were not possible to anticipate during the design of the system.

The central challenge is therefore to design systems that can maintain their security and safety properties in light of these unknowns.

2 SYSTEM MODEL

Sustainable security & safety is a desirable property of any critical system, but is particularly relevant to long-lived systems, especially those that are easily accessible to attackers. These systems are likely to be comparatively complex, consisting of multiple subsystems and depending on various external systems. Without loss of generality, we use the following terminology in this paper, which is aligned with other proposed taxonomies, such as [4]:

- **System:** a composition of multiple subsystems. The subsystems can be homogeneous (e.g., nodes in a swarm) or heterogeneous (e.g., components in an autonomous vehicle).
- **Failure:** degradation of functionality and/or performance beyond a specified threshold.
- **Error:** deviation from the correct service state of a system. Errors may lead to subsequent failures.
- **Fault:** adjudged or hypothesized cause of an error (e.g., a dormant vulnerability of the system through which adversaries may infiltrate the system, causing an error which leads to a system failure).
- **System failure:** failure of the overall system (typically with catastrophic effect).
- **Subsystem failure:** failure of an individual subsystem. The overall system may be equipped with precautions to tolerate certain subsystem failures. In this case, subsystem failures can be considered as faults in the overall system.
- **Single point of failure:** any subsystem whose failure alone results in system failure.

S3 aims to achieve the following two goals:

(1) **Avoid system failure:** The primary goal is to avoid failure of the overall system, including those originating from unforeseeable causes and future attacks. Note that avoiding system failure refers to both the safety and security properties of the system (e.g., a breach of data confidentiality or integrity could be a system failure, for certain types of systems).

(2) **Minimize overheads and costs:** Anticipating and mitigating additional threats generally increases the overheads (e.g., performance and energy) as well as the initial costs (e.g., direct costs and increased design time) of the system. In addition, higher system complexity (e.g., larger code size) may lead to increased fault rates and an increased attack surface. On the other hand, premature system failure may also have associated costs, such as downtime, maintenance costs, or penalties. Therefore, a secondary goal of S3 is to develop technologies and approaches that minimize all these potential overheads and costs.

3 CHALLENGES OF LONG-TERM OPERATION

In our S3 paradigm, we accept that we cannot know the precise nature of the attacks, faults, and changes that a long-lived system will face during its lifetime. However, we can identify and reason about the fundamental classes of events that could arise during the system's lifetime and that are relevant to the system's security and dependability. Although these events are not exclusive to long-term operation, they become more likely as the designed lifetime of the system increases. We first summarize the main classes of challenges, and then discuss each in detail in the following subsections.

Subsystem failures: In consequence of the above uncertainty about the root cause and nature of faults leading to subsystem failure, traditional solutions, such as enumerating all possible faults and devising mitigation strategies for each fault class individually, are no longer applicable. Instead, reasoning about these causes must anticipate a residual risk of unknown faults ultimately leading to subsystem failures and, treating them as faults of the system as a whole, mechanisms included that are capable of mitigating the effects of such failures at the system level. Subsystem failures could be the result of random events or deliberate attacks, possibly due to new types of attack vectors being discovered. In the most general case, any type of subsystem could fail, including hardware failures, software attacks, and failures due to incorrect specifications of these subsystems.

Requirement changes: The requirements of the system could change during its lifetime. For example, the security requirements of a system could change due to new regulations (e.g., the EU General Data Protection Regulation) or shifts in societal expectations. The expected lifetime of the system itself could even be changed subsequent to its design.

Environmental changes: The environment in which the system operates could change during the system's design lifetime. This could include changes in other interdependent systems. For example, the United States government can selectively deny access to the GPS system, which is used by autonomous vehicles for localization.

Maintainer changes: Most systems have the notion of a maintainer — the physical or logical entity that maintains the system for some part of its design lifetime. For example, in addition to the mechanical maintenance (replacing brakes, oil, etc.) Tesla vehicles regularly receive over-the-air software updates from the manufacturer.[3] For many systems, the maintainer needs to be trusted to sustain the security and safety of the maintained system. However, especially in long-lived systems, the maintainer may change (e.g., the vehicle is instead maintained by a third party service provider), which gives rise to both technical and non-technical challenges.

Ownership changes: Finally, if a system has the notion of an owner, it is possible that this owner will change over the lifetime of the system, especially if the system is long-lived. For example, vehicles are often resold, perhaps even multiple times, during their design lifetimes. Change of ownership has consequences because the owner usually has privileged access to the system. A system may also be collectively owned (e.g., an apartment block may be collectively owned by the owners of the individual apartments).

A Subsystem Failures

We anticipate subsystem failures at the hardware level, software level, and specification level. In all cases, the technical challenge is how to cope with this subsystem failure. In other words, treating the subsystem failure as a fault of the overall system, how can the system as a whole maintain safe and secure operation despite that fault?

A.1 Hardware failure. Individual subsystems may experience hardware failure during the lifetime of the system. This failure could be random, systematic, or attacker-induced and may be transient or persistent. We distinguish faults in the peripheral support systems (such as the power supply) from faults in the sensors and actuators of control systems and from faults in the processors in peripheral devices and in the main controlling units, while focusing in the following primarily on this third class of faults due to the significant higher complexity of these components. Redundant power supply infrastructures and mechanically fault tolerant actuators (e.g., with 2/3rd pressure overrides in airplane elevator actuators) are examples of solutions to address the former two classes. We further distinguish faults in the very building blocks of modern processors from faults at the architectural level, which, although being hardware, share many similarities with software faults. Clearly, fault mitigation strategies largely differ between systems that remain accessible during active maintenance phases and that can be replaced in the presence of faults in a commercially viable manner from those systems that can no longer be accessed or only at unbearable costs. For the former class, detection and diagnosis of hardware-level subsystem failures become prime objectives, followed by the initiation of the replacement process. Hardware-fault mitigation in not as accessible systems, on the other hand, has to focus on mechanisms to extend the hardware lifelime and on strategies to gracefully remove no-longer working, but left in the field, subsystems.

Extending Hardware Lifetime: Like flash memory cells, essentially all kinds of transistors wear out over time.[4] The root cause for this wear out is charges dissipating from the positively and negatively doped regions into the gate, changing the bandgap characteristics of the transistor. This causes the transistor and eventually the circuit to fail.

Emerging materials may offer a solution to loss of doping-induced charge separation. For example Silicon Nanowire reconfigurable field effect transistors (SiNW-RFETs) [36] and similar 1D-transistor structures (like Carbon nanotube or Carbon ribbon transistors) require no doping and are therefore immune to doping-related ageing or malicious wearout attacks. Instead, charges at a second reconfiguration gate define the transistor type (i.e., whether it exhibits P- or N-type behavior).

Like wear leveling, fail-over to redundantly installed circuits is one of the existing solutions to extend the lifetime of hardware circuits.[4] However, long-term sustainable security & safety requires a much more aggressive application of such solutions, possibly in addition to solutions for relocating highly sensitive functionality (like the trusted platform module) to spare hardware while irrecoverably destroying the secrets embedded in the old hardware. This gives rise to the following challenges:

A.1.(1) How can we protect spares from environmental and adversarial influences such that they remain available when they are required?

A.1.(2) How can we assert the sustainability of emerging material circuits, without at the same time giving adversaries the tools to stress and ultimately break these circuits?

Architectural Faults: At the architectural level, the same design and implementation flaws that cause software to fail apply, although to a lesser extent also at the hardware level. Let us elaborate on this using the example of hardware side channels.

[3] https://www.tesla.com/support/software-updates

[4] https://spectrum.ieee.org/semiconductors/processors/transistor-aging

Meltdown [31] and the different Spectre variants [25] recently reminded us how brittle security can be at the hardware level, in particular for highly performance-optimized systems. Micro-architectural side channels have been studied for a long time resulting in both various attacks [9, 18, 22, 26, 32, 38, 41, 53] as well as defenses [7, 10, 11] and research on software-level side channels goes all the way back to Lampson's seminal paper in 1973 [27]. The root cause of these side channels is the shared use of resources between supposedly isolated domains, e.g., the concurrent use of memory caches leads to leakage of memory access patterns potentially leaking sensitive information such as cryptographic keys. These attacks have been shown to be able to overcome the isolation between individual processes [26, 53], between processes and the kernel [21], between virtual machines [22, 32, 53], as well as the isolation of trusted execution environments (TEEs) like Intel SGX [9, 18, 38, 41]. Recently micro-architectural effects have been used in a new class of powerful attacks that allow attackers to circumvent isolation between different security domains and extract virtually all memory content across domain boundaries [25, 31, 42]. By exploiting the fact that memory access control is not enforced for CPU-internal state, e.g., during speculative execution, information can be extracted. In particular, this applies when the adversary can manipulate the CPU-internal state to have an effect on an observable side channel [31]. More generally, recently attacks exploiting hardware bugs (or undocumented behavior) from software are increasing. For instance, on ARM systems, the misconfiguration of the power management system can impact security critical functionalities in the ARM TrustZone's secure world [47]. The rowhammer bug [24] enables an adversary to flip bits in memory of another security domain, which can be triggered and exploited in various scenarios [6, 19, 42, 48], e.g., for escalating privileges from a user process to kernel privileges [42].

Defences against micro-architectural side-channel attacks and hardware bugs usually cannot be easily or efficiently implemented in software [7, 11]. While individual software defense solutions can be considered practical they usually only provide protection against individual bugs or a small class of bugs [8]. Modifications in the hardware design allow much simpler solutions with less impact on the system's performance but cannot be deployed in legacy systems [13].

The primary challenge at the hardware architectural level is therefore how can we construct containment domains such that despite failure of individual subsystems, the system as a whole can survive in a safe and secure manner or gracefully shut down without compromising security if the former is no longer possible? With side channels on the one hand and integrity threats like rowhammer on the other, this leads to the following specific challenges:

A.1.(3) How can we protect confidential information in subsystems or, if this is not possible over extended periods of time, how can we ensure confidential information is securely transferred and protected in its new location without residuals in the source subsystem revealing this information?

A.1.(4) How can we prevent one compromised hardware subsystem from compromising the integrity of another subsystem?

From what we observe today, solutions to these challenges cannot focus either on hardware or on software separately. Instead, most of the challenges we identify in this section require combined hardware and software solutions.

Graceful exclusion of faulty hardware subsystems: As in fact every physical implementation will exhibit some susceptibility to external disturbances, be they accidental or malicious, mechanisms are required to gracefully exclude faulty hardware subsystems from the overall subsystem operation. Of course, for sufficiently isolated components, solutions such as secret removal, communication prevention (e.g., to address babbeling idiots) and ultimately the shutdown of the device are possible solutions, bearing only the risk of prevention through an interposing adversary (e.g., reactivating an already shut down device). For more tightly integrated components, confinement of their behavior in the presence of external disturbances, has to start from sound models how certain circuits and logic elements react to external disturbance mechanisms, a problem which remains largely unsolved. Its solution would allow to conclusively confine the erroneous behavior of a submodule without introducing costly hardware-level isolation mechanisms (such as moats and drawbridges [20]) and without at the same time excluding the peripheral components of the malfunctioning unit, which happen to reside in the same isolation domain. The primary challenges for graceful exclusion of faulty hardware subsystems are therefore:

A.1.(5) How can we prevent adversaries from exploiting/triggering safety/security functionality of excluded components?

A.1.(6) How can we model erroneous behavior of hardware components in the presence of external disturbances?

A.1.(7) How can we construct inexpensive, fine grain isolation domains to confine such errors?

A.2 Software failure. Software attacks are constantly evolving and new vulnerabilities are discovered in software products almost every day.[5] Even worse, software vulnerabilities have become a valuable commodities traded in (black) markets.[6] Over the past years software attacks and defenses have been in a constant arms race. New defense mechanisms like control-flow integrity (CFI) [1] have been proposed to defeat different classes of attacks, such as return-oriented programming [43], and have been broken by later, more sophisticated attacks, e.g., control-flow bending [12], COOP [40], etc. It is not hard to imagine that this arms race will continue and new vulnerabilities and attack techniques will be discovered in the future. This makes it challenging to develop software systems that will remain secure in light of new attacks unknown at the time of development/deployment. Additionally, attack vectors that are closed in desktop and server systems, like code injection attacks, which can be prevented with data execution prevention (DEP),[7] are still a threat for many other classes of devices, in particular embedded systems. This gives rise to the following challenges:

A.2.(1) How to design systems that can detect and isolate software subsystem failures?

[5] https://techtalk.gfi.com/2015s-mvps-the-most-vulnerable-players/

[6] http://zerodium.com/

[7] https://support.microsoft.com/en-gbus/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in

A.2.(2) How to transfer software attack mitigation strategies between domains (e.g., PC to embedded)?

A.3 Subsystem compromise. The constant discovery of vulnerabilities in software as well as the fact that over time new attacks methods are developed lead to the situation that we have to expect subsystems to get compromised at some point in the systems life time. Unlike random failures targeted attacks propagate through multiple subsystems by exploiting control over one subsystem to gain control over another. Also, the adversary can ensure that the behavior (e.g., interactions with other subsystems) of a compromised subsystem remains correct to hide the attack make the detection of stealthy attacks particularly challenging. Additionally, if a subsystem is compromised all confidentiality guarantees are voided, i.e., if the subsystem has access to a secret like a cryptographic key the adversary will extract that key. Hence, even if the subsystem can be reset or even updated the key's confidentiality remains compromised and the subsystem needs to be re-provisioned. This raises the following technical challenges:

A.3.(1) How to recover a system when *multiple* subsystems are compromised?
A.3.(2) How to detect that a subsystem is compromised by a stealthy adversary?
A.3.(3) How to react to the detection of a (potentially) compromised subsystem?
A.3.(4) How to prevent the leakage of sensitive information from a compromised subsystem?
A.3.(5) How to securely re-provision a subsystem after all its secrets have been leaked?

A.4 Specification failure. Even if all defects in the software and hardware implementations could be removed, e.g., by formally verifying all parts of the implementation, the system may still fail due to faults in the specification. The primary challenge with faults at the specification level is to rule out common-mode problems, which carry the potential to affect most, if not all implementations of a given specification. Moreover, as we discuss in the later subsections, a change of requirements or environment may also affect the specifications and the validity of assumptions upon which they depend.

An important subclass of specification failure is *assumption failure*, which encompasses all cases in which the assumptions used in the specifications cease to hold. For example, the correctness of cryptographic algorithms and protocols often hinges on the assumption of certain problems being computationally *hard*. However, this may no longer hold as new computational capabilities (e.g., quantum computers) or new mathematical insights come to light. At worst, algorithms based on such assumptions could constitute a single point of failure for the system, causing a full system failure if not mitigated before adversarial exploitation. For example, failure of one or more of the cryptographic algorithms used in systems like Bitcoin could have catastrophic consequences on the system [15].

Specification-level failures give rise to the following technical challenges:

A.4.(1) How to design subsystems that may fail at the implementation, but not at the specification level (and at what costs)?
A.4.(2) If specification faults are inevitable, how to design systems in which subsystems can follow different specifications whilst providing the same functionality, in order to benefit from diversity of specifications and assumptions?
A.4.(3) How to recover when one of the essential systems has been broken due to a specification error (e.g., how to recover from a compromised cryptographic subsystem)?

In addition to the hardware, software, and specification failures described above, a subsystem could also fail for other reasons, including changes to third-party dependencies (see Section C) or the subsystem becoming unmaintained (see Section D).

B Requirement Changes

B.1 Regulatory changes. Even though a system is compliant with current regulations at the time of design, it is possible that the regulatory (legal) framework will change during the lifetime of the system. In some cases, pre-existing systems might be explicitly excluded from such changes. However, if the changes concern safety or security, it is likely that they will be applied to all systems, including those already deployed. For example, the recent EU General Data Protection Regulation (GDPR) applies to all systems, irrespective of when they were designed. This gives rise to the following technical challenges:

B.1.(1) How to retroactively change the designed security, privacy, and/or safety guarantees of a system?
B.1.(2) How to prove that an already-deployed system complies with new regulations?

B.2 User expectation changes. More subtly than an explicit change in regulation, the expectations of the users may change with regard to security and safety. For example, improvements in braking technology on cars have led to shorter stopping distances, which has arguably changed the expectation of drivers and pedestrians over time. Although this example is a limitation of the system's hardware, Tesla has recently demonstrated the ability to improve the braking performance of vehicles with only a software update.[8] In addition to the challenges in dealing with regulatory changes, this gives rise to the following technical challenge:

B.2.(1) How can a system be extended and adapted to meet new expectations after deployment?
B.2.(2) How to demonstrate to users and other stakeholders that an already-deployed system meets their new expectations?

B.3 Design lifetime changes. One important category of a change in user expectation is a change in the intended or expected design lifetime of the system. For example, the Mars rovers Spirit and Opportunity have been laid out for a guaranteed mission time of 90 days on Mars (approximately 92 days on Earth), which both rovers exceeded significantly. While Spirit was abandoned after 7 years of operation, Opportunity was active for almost 15 years, i.e., almost 60 times the initially planned mission duration. This is critical because the extension of a system's design lifetime may exacerbate all the challenges described above. This gives rise to the following technical challenges:

[8] http://fortune.com/2018/05/27/tesla-model-3-braking-update/

B.3.(1) How to determine whether a deployed system will retain its safety and security guarantees for an even longer lifetime?
B.3.(2) How to further extend the safety and security guarantees of a deployed system?

C Environmental changes

C.1 New threat vectors. Despite extensive efforts in identifying and mitigating covert and side channels, many of the recent hardware-level vulnerabilities (like Meltdown [31], Spectre [25] and Rowhammer [42]) and, in particular, the ease of exploiting them, came as a surprise. In the future, we expect many more of these surprises to happen as researchers, but also less honest individuals (putting dishonest researchers into the latter class) continue to investigate the security and safety of systems and as the environments in which systems execute change. The main challenges are therefore:

C.1.(1) How to tolerate failure of subsystems due to unforeseeable threats?
C.1.(2) How to avoid single points of failure that could be susceptible to unforeseen threats?
C.1.(3) How to improve the modeling of couplings and dependencies between subsystems such that the space of "unforeseeable" threats can be minimized?

Naturally, one may take the position that countering the unforeseeable is a futile task due to the unknown nature of what may happen, and this position may turn out to be valid. Nevertheless, we believe measures can be taken to improve the tolerance despite unknowns, including anticipating unlikely incidents without disqualifying them due to their rareness and, most importantly, by not addressing safety and security separately. Safety vulnerabilities, which are rarely triggered by random events and natural causes, may well become a threat when exploited by intentional adversaries.

C.2 Unilateral system/service changes. Any non-trivial system is likely to depend on various external third-party systems or services. Failures or unilateral changes in these may have an impact on a long-lived system. For example, modern standards-compliant web servers are no longer permitted to accept SSLv3 connections for security reasons.[9] Any client systems still relying on this protocol would thus be unable to connect. There is a clear security reason for this change, and it is beneficial to force clients to upgrade, but it assumes that all clients have the ability to upgrade. This change was also forewarned well in advance. However, this may not be the case for all changes, and thus systems must be able to adapt sufficiently quickly. This gives rise to the following technical challenge:

C.2.(1) How to design systems such that any third-party services on which they depend can be safely and securely changed?
C.2.(2) How can a system handle unilateral changes of (external) systems or services?

C.3 Third-party system/service fails. A more severe case is when a third-party service fails, which often takes place without prior warning. In the simplest case a service is not reachable anymore due to connectivity loss, i.e., failure of the service's network connection, the system's network connection, or a failure in some intermediate network infrastructure like the Internet. Similar, many secure systems depend on the public key infrastructure (PKI), which is based on trust in Certificate Authorities (CAs). However, evidence has shown that CAs can *fail* by losing this trust (e.g., as the result of key compromise).[10] Furthermore, third-party systems or services are usually beyond the control of the maintainer, for example, the United States government can selectively deny access to the GPS system, which essentially constitutes a failure of the system. In addition to the challenge above, this gives rise to the following technical challenge:

C.3.(1) How can a system handle the failure or unavailability of external services?
C.3.(2) How to design systems such that any third-party services on which they depend can be safely and securely changed *after they have already failed*?

C.4 Maintenance resource becomes unavailable. During the lifetime of the system, the tools or resources required to maintain the system could become unavailable. This applies to physical tools, software tools (e.g., compilers), and human resources. In a long-lived system, it may become increasingly difficult to find software developers with the necessary skills to update/maintain older technologies. For example, it is estimated that USD 3 trillion in daily commerce still flows through systems running COBOL,[11] a language rarely taught today. The very high cost to replace these systems means that maintainers are forced to rely on a shrinking pool of COBOL developers to fix issues and update the systems. This gives rise to the following technical challenges:

C.4.(1) How to identify all maintenance resources required by a system?
C.4.(2) How to maximize the *maintenance lifetime* of a system whilst minimizing cost?
C.4.(3) How to continue maintaining a system when a required resource becomes completely unavailable?

D Maintainer Changes

For various reasons the maintainer of the system might need to be changed. For example, the original maintainer could stop providing updates. However, allowing someone else to take over the role of the maintainer raises various security and dependability challenges.

D.1 Implementing a change of maintainer. Firstly there is the question of who gets to decide when to change maintainer, and who the new maintainer should be. On one hand, the owner of the system has an interest in minimizing cost and keeping the system running, which may not be possible if the original maintainer stops providing updates or goes out of business. On the other hand, the original maintainer (e.g., the car manufacturer) may still have ongoing legal obligations/liabilities for the system during this time, or may suffer a loss of reputation if the system fails due to improper maintenance. When a change of maintainer takes place, there is also a need to securely convey this change to the system. If this functionality is

[9] https://tools.ietf.org/html/rfc7568

[10] https://threatpost.com/final-report-diginotar-hack-shows-total-compromise-ca-servers-103112/77170/

[11] https://www.reuters.com/article/us-usa-banks-cobol/banks-scramble-to-fix-old-systems-as-it-cowboys-ride-into-sunset-idUSKBN17C0D8

not protected, it can be abused by an adversary to take over the role of maintainer. This gives rise to the following technical challenge:

D.1.(1) How to ensure that a system remains secure and safe even under a new maintainer?
D.1.(2) How to securely inform the system that a change of maintainer has taken place?

D.2 System becomes unmaintained. For various reasons the system could become unmaintained during its lifetime. For example, the manufacturer could stop providing updates, or the owner could neglect to update the system (e.g., an owner who blocks software updates). In any case, the system needs to detect that maintenance is not provided any more, e.g., by notification from previous maintainer or automatic detection. When loss of maintenance is detected, how should a system be designed to maximize its effective lifetime? An unmaintained system is generally less able to respond to attacks and needs a strategy to react to this situation (e.g., shutdown or allow anyone to take over the maintainer role). This gives rise to the following technical challenges:

D.2.(1) How can a system decide that it is no longer maintained?
D.2.(2) How should an unmaintained system behave?

E Ownership Changes

In many scenarios, proof-of-presence is used to set up credentials for subsequent proof-of-possession authentication, e.g., on a typical desktop system the owner is allowed to install new keys into the UEFI secure boot system when physically present in front of the computer. This allows the owner to boot arbitrary (potentially malicious) operating systems and software. When these keys remain on the system after ownership changes they can serve the old owner as a gateway to attack the system. On the other hand, sensitive data of an old owner can be targeted by a new system owner, e.g., re-sold smartphones regularly contain sensitive information of the previous owner.[12]

Transitioning from proof-of-presence to more direct proof of possession schemes (such as smardcard-based authentication) avoids some of the above problems, but introduces others (e.g., loss or theft of authenticating tokens). In the following, we distinguish cooperative from non-cooperative ownership changes.

E.1 Cooperative ownership changes. The system may change ownership between two cooperative entities (e.g., a car is sold or a rental car is returned). However, although the entities cooperate to perform this change, they may still be adversarial from each other's perspective. For example, the system my contain valuable information from the previous owner (e.g., personal information, access credentials, or payment details) that the new owner may attempt to learn. Conversely, the previous owner may attempt to retain access to the system either to obtain data about the new owner (e.g., snoop on the new owner), or even maliciously control the system. This gives rise to the following technical challenges:

E.1.(1) How to securely inform the system about the change in ownership, without opening a potential attack vector?
E.1.(2) How to erase sensitive data during transfer of ownership, without allowing the previous owner to later erase usage/data of the new owner?

E.2 Non-cooperative ownership change. In addition to the challenges of a cooperative ownership change, there may be circumstances in which the current owner is unwilling or unable to participate in any ownership change protocol (e.g., a car is impounded or repossessed from the owner and sold to a new owner). It is assumed that there is still some other legal basis to legitimize the change of ownership (e.g., contract or legal regulation), but this needs to be reflected in the technical design of the system. This is challenging because the non-technical basis for such an ownership change may itself change over the lifetime of the system (e.g., the law might change, see section 3.5.1). This gives rise to the following technical challenge:

E.2.(1) How to automatically detect non-cooperative ownership change?
E.2.(2) How to erase sensitive data after loss of ownership, without allowing the previous owner to erase usage/data of the new owner?

4 TECHNICAL IMPLICATIONS

The challenges identified in Section 3 lead to a number of technical implications, which we highlight in the following before presenting more concrete proposals to address sustainable security & safety in the next section. In particular, the realization that any subsystem may fail, especially given currently unforeseeable threat vectors, and the realization that subsystem failure is a fault in the overall system, demands a principled treatment of faults.

Although safety and security independently developed solutions to characterize the risk associated with faults, we observe a gap to be closed, not only for automotive systems [30], but also in any ICT system where risks are assessed based on the probability of occurrence. Once exposed to adversaries, it is no longer sufficient to capture the likelihood of natural causes coming together as probability distributions. Instead, probabilistic risk assessment must take into consideration the incentives, luck, and intents of increasingly well-equipped and highly skilled hacking teams. The conclusion may well be high risk probabilities, in particular when irrational hackers (such as cyberterrorists) are taken into account. Also, naively applying techniques, principles, and paradigms used in isolation in either safety or security will not solve the whole problem, and worse, may interfere with each other. Moreover, to reach sustainability, any such technique must withstand long-term operation, significantly exceeding that of recent approaches to resilience.

For example, executing a service in a triple modular redundant fashion increases tolerance against accidental faults in one of the replicas. However, it does not help to protect a secret passed to the individual replicas by encrypting this secret for each replica and with the replica's key. Once the key of a single compromised replica is extracted, confidentiality is breached and the secret revealed. Further, if a replica fails due to an attack rather than a random fault, replaying the attack after resetting this replica will often recreate this failure and eventually exhaust the healthy majority.

[12] http://www.govtech.com/security/Massive-Volume-of-Sensitive-Data-on.html

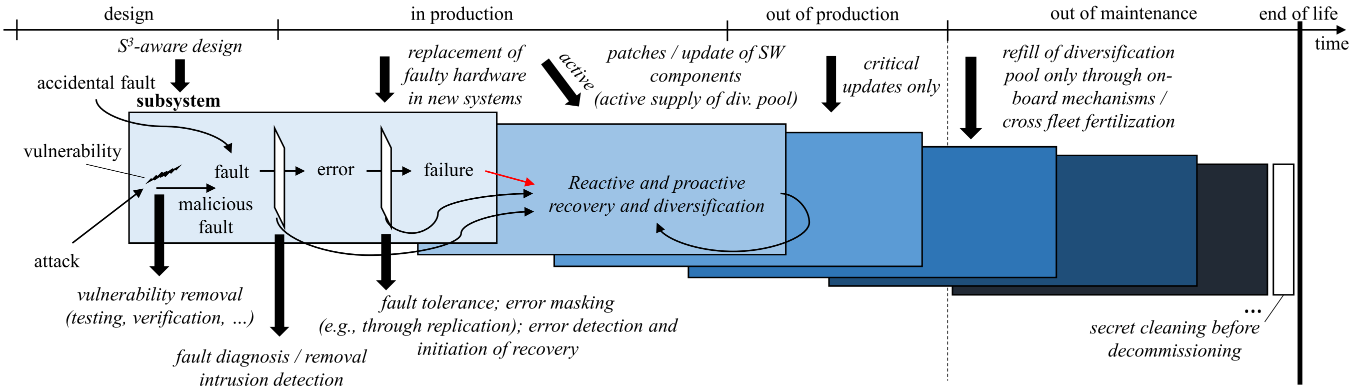
design
in production
out of production
out of maintenance
end of life
time
S^3-aware design
accidental fault
subsystem
vulnerability
fault
malicious fault
attack
error
failure
replacement of faulty hardware in new systems
active
patches / update of SW components (active supply of div. pool)
critical updates only
refill of diversification pool only through on-board mechanisms / cross fleet fertilization
Reactive and proactive recovery and diversification
vulnerability removal (testing, verification, ...)
fault diagnosis / removal intrusion detection
fault tolerance; error masking (e.g., through replication); error detection and initiation of recovery
...
secret cleaning before decommissioning

Figure 2: Means to attain dependability and security.

To approach sustainable security & safety, we need a common descriptor for the root-cause of problems in both fields — *faults* — and a principled approach to address them [49, 50]. As already pointed out in [4], faults can be accidental, for example, due to natural causes, following a stochastic distribution and hence justifying the argument brought forward in safety cases that the residual risk of a combination of rare events is within a threshold of accepted risks. However, faults can also be malicious, caused by intentional adversaries exploiting reachable vulnerabilities to penetrate the system and then having an up to 100% chance of triggering the combination of rare events that may lead to catastrophe. Faults can assume several facets, and the properties affected are the union of those both fields and are concerned with: reliability, availability, maintainability, integrity, confidentiality, etc.

Returning to the above example, treating a confidentiality breach as a fault, it becomes evident why encrypting the whole secret for each replica individually constitutes a single point of failure. It also gives rise to possible solutions, such as secret sharing [44], but further research is required to ensure long-term secrecy for those application fields that require this, e.g., simple secret sharing schemes do not tolerate a mobile adversary that subsequently compromises and releases one replica at a time.

In the long run, failure of individual subsystems within the lifetime of the overall system becomes an expected occurrence. We therefore distinguish normal failure (e.g., ageing causing unavailability of a subsystem's functionality) from catastrophic failure (causing unintended system behavior), and devise means to attain dependability and security despite both.

4.1 S3 Lifecycle

Figure 2 sketches the path towards attaining dependability and security, and principled methods that can be applied over the lifetime of the system to obtain sustainable security & safety. Faults in subsystems manifest from attacks exploiting reachable vulnerabilities in the specification and implementation of the subsystem or from accidental causes. Without further provisioning, faults may manifest in errors, which may ultimately lead to failure of the subsystem. Here we take only the view of a single subsystem, which may well fail without inevitably implying system failure. More precisely, when extending the picture in Figure 2 to the overall system, subsystem failures manifest as system faults, which must be caught at the latest at the fault tolerance step. It may well be too late to recover the system as a whole after a failure has manifested, since this failure may already have compromised security or caused a catastrophe (indicated by the red arrow).

4.1.1 Design. Before deployment, the most important steps towards sustainable security & safety involve preparing the system as a whole for the consequences of long-term operation. In the next section, we sketch early insights what such a design may involve. Equally important is to reduce the number of vulnerabilities in the system, to raise the bar for adversaries. Fault and vulnerability prevention starts from a rigorous design and implementation, supported for example by advanced testing techniques such as fuzzing [16, 17, 37, 39]. However, we also acknowledge the limitations of these approaches and consequently the infeasibility of zero-defect subsystems once they exceed a certain complexity, in particular in the light of unforeseeable threats. For this reason, intrusion detection and fault diagnosis and fault and vulnerability removal starts after the system goes into production,[13] which also is imperfect in detecting only a subset of intrusions and rarely those following unknown patterns while remaining within the operational perimeter of the subsystem. It is important to notice that despite the principled imperfection of fault, vulnerability and attack prevention, detection and removal techniques, they play an important role in increasing the time adversaries need to infiltrate the system and in assessing the current threat level, the system is exposed to.

4.1.2 In Production. While in production, replacement of faulty hardware components is still possible, by providing replacement parts to those systems that are already shipped and by not shipping new systems with known faulty parts. Software updates remain possible during the whole time when the system remains under maintenance, although development teams may already have been relocated to different projects after the production phase.

Fault tolerance, that is, a subsystem's ability to gracefully reduce its functionality to a non-harmful subset (e.g., by crashing if replicas start to deviate) or to mask errors (e.g., by outvoting

[13] We consider design-time penetration testing as part of the vulnerability removal process.

the behavior of compromised or malicious replicas behind a majority of healthy replicas, operating in consensus) forms the last line of defense before the subsystem failure manifests as a fault. The essential assumption for replication to catch errors not already captured by the fault and error removal stages is fault-independence of all replicas, which is also known as absence of common mode faults. Undetected common mode faults bear the risks of allowing compromise of all replicas simultaneously, which gives adversaries the ability to overpower the fault tolerance mechanisms. Crucial building blocks for replication-based fault tolerance are therefore the rejuvenation of replicas to repair already compromised replicas and in turn maintain over time the required threshold of healthy replicas (at least one for detecting and crashing and at least $f + 1$ to mask up to f faults) and diversification to avoid replicas from failing simultaneously, and to cancel adversarial knowledge how replicas can be compromised.

4.1.3 Out of Production. As long as the system is maintained, the rejuvenation and diversification infrastructure for the system may rely on an external supply of patches, updates and new, sufficiently diverse variants for the pool of diverse subsystem images.

4.1.4 Out of Maintenance. Once the system falls out of active maintenance, replenishment of this pool is limited to on-board diversification mechanisms (such as binary obfuscation[14]) or through fleet-wide cross-fertilization, by exchanging diagnosis data between systems of the same kind, which allows one system to learn from the intrusions that happened to its peers. The final state before the end of life of the system is a reliable cleaning step of all secrets that remain until this time, followed by a graceful shutdown of the system.

5 DESIGN PRINCIPLES

Figure 3 sketches one potential architecture for sustainable security & safety and shows the abstract view of a sustainable systems and its connections/relations with its surroundings. A system interacts (possibly in different phases of its life cycle) with different stakeholders. The stakeholders (or a subset thereof) usually define the applications and objectives of a systems, e.g., the manufacturer and developer define the primary/intended functionality of a systems. The system's intended applications or objectives, as well as external factors such legal frameworks, define the overall requirements a system must fulfill.

The center of Figure 3 shows the overall system that can be composed from multiple subsystems, each of which is a combination of multiple components. Components of a subsystem are connected with one another. Backup components (marked in grey) are required to achieve fault tolerance, i.e., if one component of a subsystem fails it can be (automatically) replaced by a backup component.[15]

The subsystems are connected and interact via defined interfaces (shown as tubes). As long as the interfaces and the individual subsystems are fault tolerant, the overall system can inherit this property.[16]

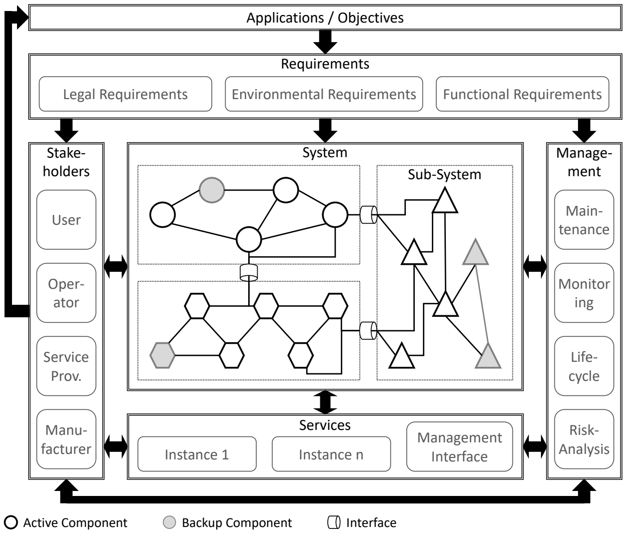

Figure 3: Abstract view of a sustainable system and its surroundings.

Sustainable systems usually do not operate in isolation throughout their life-time. They often rely on external services, e.g., cloud services to execute computation intensive tasks. If these services are critical for the operation of the system, the resiliency of these services is relevant, e.g., multiple instances of the service must be available, and the connection to the external service must be reliable, as well.

The system's management includes technical management as well as abstractly controlling the system and its operation. This has to cover both the management of the system itself and the external services upon which the system relies. The management can be performed by a single entity (stakeholder) or distributed among different stakeholders, e.g., the device manufacturer provides software updates for the system while the user configures and monitors the system.

Based on the challenges discussed in the preceding section, we can already deduce certain architectural requirements and design principles for achieving S3. In the following, we discuss these principles and identify existing work that could serve as the initial building blocks.

5.1 Well-defined Components and Isolation

Complex services are usually provided by complex systems. To ensure that the complexity can be handled, breaking down the functionality into well-defined components is essential. Furthermore, strongly isolating components and limiting interaction to well-defined and constrained interfaces helps limiting error propagation. A particular challenge is that isolation of two subsystems

[14] https://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-sean_taylor-binary_obfuscation.pdf

[15] Backup components can also be online/active to reduce the load per component as long as not failure has occurred.

[16] If an individual subsystem cannot provide the required fault tolerance level, the overall system requires redundancy with regard to that subsystem.

will usually require sufficient isolation of all underlying subsystems including the hardware.

A related design principle is to aim at separating services of different criticality into components. The benefit is that critical components will justify higher security and dependability investments while uncritical components can be implemented at lower cost.

5.2 Avoid Single Points of Failure

For systems in long-term operation, it must be expected that any subsystem could fail, especially when exposed to unforeseen or unforeseeable attacks or changes. Subsystem failure could affect either the functionality provided by individual subsystems, hence impacting the safety of the components they control, or the security of the subsystems. Anticipating this possibility, it is clear that no single subsystem can be responsible for the safety and security of the system as a whole. In the face of unknown threats this entails the use of diversity to prevent common cause failures rooted in a single unforeseen cause (see Section 5.9). For the same reason, a multi-level approach for safety and security, with overlapping coverages, is beneficial.

5.3 Multiple Lines of Defense

From past experiences we know that individual defenses can be overcome by an adversary, hence, relaying on a single defense mechanism represents a single point of failure with respect to the goal of protecting the system from attacks. Hence, multiple lines of defense are needed to protect the system even if an individual defense mechanism can be circumvented by an adversary. For instance, multiple cryptographic algorithms can be combined in such a way that an adversary has to break different cryptographic systems to get access to critical information. However, the diversity of the used algorithm need to be sufficient, i.e., they should be based on different mathematical problems that cannot be easily be translated into one another. If prime decomposition is efficiently solvable also the direct logarithm problem can be efficiently solved and therefore combining two algorithms based on these problems might not provide the desired protection. Similarly defense mechanisms, for instance against run-time attacks, should be combined to increase the system's security. Control-flow integrity (CFI) [1], for instance, has been attacked by full-function reuse attacks [12, 40] while randomization approaches suffer from information leakage [45]. Combining both can increase the security, e.g., by relying on randomization to mitigate full-function reuse attacks to undermine CFI.

5.4 Long-term Secrets and Confidentiality

Ensuring the confidentiality of data over long periods of time is very challenging since data that once leaked cannot become confidential again. For instance, if the mathematical problem a cryptographic algorithm is based on gets efficiently solvable (like the prime decomposition for RSA) secret keys will not be confidential anymore. Even replacing a revealed key with a new key will not solve the problem as the new key can be retrieved by the adversary the same way as the previous key. Similarly, if a subsystem handling secret information has failed and has leaked the secret information, e.g., because an adversary has compromised the subsystem to manipulate its behavior to output the secret, then a new secret can be leaked in a similar way as long as the subsystem is not updated. To prevent the leakage of secret information their use should be minimized. Additionally, a single subsystem should not be allowed to access a secret as a whole, as this subsystem would become a single point of failure with respect to the secret information's confidentiality.

5.5 Robust Input Handling

A (sub)system should not make assumptions with respect to the inputs it expects. This holds true for external inputs as well as for inputs between subsystems. Malicious inputs can, on one hand, lead to faults and failures in the system, like buffer overflows, which can crash the system or could even be exploited by an adversary to gain control over the system. On the other hand, "unexpected" inputs can lead to undesired behavior of a system, e.g., in machine learning based systems. Hence, a robust system should incorporate two aspects: (a) the system should cope well with noisy and uncertain real-world input data, and (b) express its own knowledge and uncertainty of a proposed output even with unforeseen input [35]. Another aspect of robustness requires the decision-making system to be robust against adversarial tampering with inputs.

5.6 Contain Subsystem Failure

An immediate conclusion from this requirement to tolerate arbitrary failures of individual subsystems is the need to confine each subsystem into an execution environment that acts as fault containment domain and in which the subsystem may be rejuvenated to re-establish the functionality it provides. Ideally, sustainable secure systems should be constructed such that compromise of any minority subset of components reveals no security sensitive information. In particular, no such information leak should happen from one subsystem to another (unless explicitly authorized) as subsystems may hold secrets from multiple stakeholders. Unfortunately, side channels and flaws in individual subsystems make this very challenging for all but the most simple or robust system components.

Rigorous interface analysis, including out-of-spec behavior, as well as comprehensive modeling of potential coupling mechanisms and their effects are the most prominent remedies here.

5.7 Subsystem Updates

Any subsystem could fail, e.g., due to an attack, a random fault, or because a subsystem or mechanism has reached the end of its lifetime before the end of the overall system's lifetime. Therefore, it is critical to design the system such that individual subsystems can be updated during the operational lifetime of the system. In order to facilitate this, the behavior of each subsystem must be fully specified, and any updates/replacements must be checked against this specification. For example, in the case of hardware changes, more modern hardware may operate at higher clock frequencies, which may need to be downscaled to interoperate with older subsystems. In the case of software systems, this means anticipating the need for software updates, and ensuring that these updates can be applied safely and securely, and not give rise to new attack vectors. In the most extreme case, if a subsystem has already failed, it should ideally be possible to restore this subsystem to a correct state (i.e., *rejuvenated*).

For the hardware platform this favors the use of reconfigurable hardware like soft-cores and FPGA platforms rather than custom ASICs. Although the latter can be custom-tailored and hence exhibit lower cost, complexity and area, the reconfigurability of the former is a significant advantage for long-living systems.

5.8 Replicate to Tolerate Subsystem Failure

At the application level, if the functionality provided by the failing subsystem cannot be compensated by lower-level components (possibly at a different quality of service), the subsystem must be replicated such that failure of a subset of these replicas can be masked behind the majority of still healthy replicas operating in consensus. Considering autonomous driving as a use case, route planning is an example of a subsystem, which may fail without consequence to the safety requirement for not crashing into obstacles, provided an independent collision detection system detects all obstacles and causes the car to stop. Likewise, the collision detection system itself must not fail and its components must therefore be replicated to tolerate failure in a subset of its components.

Going down the software stack, the same principled mechanisms apply for replicating, diversifying and rejuvenating replicas, respectively for relying on fall-back functionality if the subsystem is not essential or its functionality can be provided (at least partially) by the fall-back subsystems. However, the infrastructure available for these mechanisms decreases and ultimately ceases to exist when reaching the lowest software levels (i.e., the operating system kernel or firmware).

The costs of replication can be partially avoided for those components for which replacement or at least fall-back functionality is present. These components still require rejuvenation, but fall-back components ensure safety during replacement.

If recovery can be accomplished safely within a time shorter than the natural "fault tolerance time" of the application, replication may be avoided as well. To leverage that, fast recovery must be accomplished (avoiding complete reset, e.g.), and in order to make sure recovery is possible from all error states, self-stabilization is beneficial.

5.9 Diversify Nodes and Components

From distributed systems research we know that simple replicated execution is not enough to evade attacks [5]. Instead, replicas must be sufficiently diverse to prevent accidental faults from manifesting as failures in all replicas and to prevent adversaries from being able to simultaneously intrude into all replicas (i.e., without sequential effort). Moreover, replicas must be rejuvenated to establish fresh and sufficiently diverse instances faster than adversaries will be able to compromise them [46].

Anticipating logical or implementation level compromise, Völp et al. [52] sketch a first architectural solution for maintaining security despite compromise of encryption algorithms. Rather than relying on a single such algorithm, secret sharing techniques [44] are used in combination with classical ciphers such that no single cipher encrypts the whole secret. Hence, compromise only partially reveals information and in such a way that reconstructing the secret remains difficult. Again rejuvenation and in consequence reencryption of the secret maintains security over extended periods of time.

5.10 Enable Relocation

Hardware, in particular non-reconfigurable hardware, must follow a different strategy to evade faulty subsystems. Rather than rejuvenating, which would imply reconfiguration possibilities (i.e. only available on FPGAs or similar circuits), the software must be relocated from permanently damaged hardware subsystems to healthy spares, deactivating faulty circuits or using them in a way where faults cannot manifest in subsystem failure. One essential requirement for such a relocation is the reliable cleaning of platform components from secrets held on behalf of a stakeholder, in particular if the interests of this stakeholder in a subsystem changes. In particular when relocating functionality from broken to healthy components, such cleaning must still work reliably or the component cannot be reused for other functionality and information extraction must be prevented.

5.11 Adaptive Systems

The security and safety of the complex system is dependent on several uncertainties which make it more challenging to design secure (sub)systems during the design stage. Therefore, these (sub)systems must be intelligent enough to adapt the uncertainties and new conditions during the runtime. In particular, (sub)systems should be able to detect incidents like faults and failures, and react to them, i.e., adapt the system. In case of an incident is detected the system should adapt, for instance by enabling active protection mechanism, rejuvenation, reconfiguration, as well as (guaranteed) recovery from the incident.

Adaption and reconfiguration is relevant in order to stay ahead of adversaries, but it also becomes necessary when the environment changes in which the system operates or when the interest of stakeholders in individual subsystems changes. For example, if an autonomous car is sold, the old owner's personalization needs to be reset in order to reinitialize the car for the new owner.

5.12 Minimize Assumptions

With the envisioned long life time and unforeseeable changes ahead, assumptions that seem plausible at the time of design may no more hold during later phases of use. This may be due to requirement changes or environmental changes, as outlined in Section B and Section C, respectively. Also, attackers may find new ways of violating assumptions and thus defeat safety or security measures that rely on them, or open coverage holes. For these reasons, assumptions, although often substantially simplifying implementation, need to be minimized, as they weaken the system's robustness. Examples for such assumptions are the single-fault assumption, the already mentioned "hardness" of problems for cryptography, synchrony between components, interface design considering a limited space of input behaviors only, etc.

5.13 Simplicity and Verifiability

Most existing systems suffer from high (sub)system complexity. So simplicity is definitely a feature, and with the integration of novel measures for enhancing safety and security, the right cost trade-off must be found. A thorough analysis of all possible behaviors under all perceivable faults, ideally formal, is possible only for very small and simple entities. To perform such verification and validation, an accurate but abstract model of (sub)systems is specified and then validated/verified based on the defined security/functional properties under the design/environmental constraints. While due to performance demands it is out of reach to keep the whole system that simple, the trusted core typically required in fault-tolerant and secure architectures can be kept sufficiently small. Such a small and simple unit is furthermore less likely to suffer from unforeseen changes and threats.

6 RESEARCH DIRECTIONS

To tackle all challenges identified in the previous sections and enable systems that fulfill the necessary design principles research in various areas has to be advanced beyond the current state. Research directions and approaches for some of the most pressing challenges to advance towards the vision of sustainable secure and safe systems are listed below.

6.1 Methodology for Building S3 Systems

In the previous sections we have sketched how architectural hybridization lends a pathway to the construction of sustainable safe and secure (S3) systems. Essentially, by solving the hard problem of simultaneously achieving safe, secure, and timely control of the assets the system is entrusted with (cyber-only as well as cyber-physical), we allow it to endure both faults and intrusions within the normal operation envelope, as well as to gracefully reduce its functionality to fail-safe or even fail-operational modes, in the event of extreme hazards (accidental or intentional). Still, principled and paradigmatic research is required to turn this approach into a methodology for the systematic construction of S3 systems.

Functionality decomposition methods should inspire the definition of hybridisation boundaries between ultimately trusted and less trusted subsystems. Adequate isolation principles should confer the necessary level of trustworthiness to substantiate that trust, namely, through spacial isolation across several sub-systems, or locally, through combinations of physical and logical isolation enforcing trustworthy code execution (Section 6.2).

The assumed effectiveness of countermeasures (e.g., error detection and reaction mechanisms Section 6.3) and the risk of failure, must be assessed and validated on the basis of appropriate system models, not only of the intended behavior, but more importantly also of the behavior that can realistically be expected in the presence of faults and compromise (Section 6.5), to obtain arguments for the confidence placed in the underlying system assumptions.

Last but not least, architectural hybridisation hinging on systems having loci ('hybrids') with distinct system and fault assumptions, the availability of, and trust on, such hybrids, and on their interplay with the payload system, becomes of paramount importance. Such functionality must thus be implemented and hardened to the highest standards possible. This includes not only designing and verifying hardware primitives providing ultra-reliable services locally, but also devising techniques to algorithmically build high trust from a majority of non-perfect but good enough instances, operating in consensus (Section 6.4 very naturally assisted by the above-mentioned hardware primitives).

6.2 Local Hybrid (Trusted) Code Execution

Many application scenarios support physical separation as isolation mechanism to confine failures in untrusted components and prevent them from causing faults in other trusted or untrusted components. For example, hybridization approaches such as MinBFT [51] rely on a physically separated, trusted-trustworthy component — the USIG — to enforce sequentiality and monotonicity of messages and in turn prevent equivocation in MinBFT's leader-based consensus protocol. Implementation options include utilizing specifically hardened TPMs, holding the monotonic counter and the keys for generating authentic message signatures that are required by the USIG.

However, the setting of this hybrid solution is limited to the processing of cyber assets only and conveys no protection against secret leakage since no further precautions are installed to prevent individual faulty replicas from interacting with their environment. In this case, spacial separation is no longer a viable option, due to bypassing possibilities or due to increased requirements on the untrusted components to not cause unsafe or insecure behavior through direct interaction with critical system components (e.g., the actuators impacting the physical environment of the cyber system). These other application scenarios require further research on provisioning and enforcing distinct trust assumptions in software components that coexist on the same system. Naturally, trusted execution environments such as Intel SGX and ARM Trustzone form an important first step in this direction. However, they do not yet support all the properties S3 systems must provide and often the complexity of their implementation (e.g., having to rely on the correctness of the entire core and its firmware) reduces the coverage of their trustworthiness assumption.

In this research line, we will therefore focus on techniques to enable locally, within the same multi- and manycore, system hard- and software components with distinguished sets of trust assumptions to enable local hybridization, for example, but not exclusively, for supervising actuation of controlled physical systems.

6.3 Building Systems with Sustainable Security & Safety

The sound and robust implementation of complex software systems is not well understood in practice and we are far away from actually building systems that provide sustainable security & safety. As discussed earlier, new types of (software) attacks are constantly evolving and there is a constant stream of new vulnerabilities that are discovered. The experiences of the last two decades have shown that relying on security tools and developer training to avoid such vulnerabilities has substantial limitations. Especially in situations in which the complexity of the system exceeds a certain threshold, neither approach is sufficient to reasonably guarantee the absence of security vulnerabilities. A prime example is the *Heartbleed* vulnerability in the crypto library OpenSSL that was only detected several years after it was introduced by a programmer.

To improve this situation in a fundamental way, we need to investigate novel methods for securing computer systems at all levels of their development life cycle such that they can deal with software failures and subsystem compromises. Such methods need to take all relevant user groups – ranging from system designers through developers to end users – into account, all of them need to apply both security and safety principles in a reliable and robust way. For example, we need to investigate secure application programming interface (API) design, secure compiler chains that eliminate today's low-level vulnerabilities, and sound mechanisms to develop secure components for complex applications. Moreover, we will develop mechanisms that enable the secure and safe integration of (potentially untrusted) third-party components into an existing system. To achieve this goal, we need to design key components in a robust way that will ensure the security characteristics for the operation of the system over its entire lifetime. Such system architectures must enable a dynamic, gradual adaptation of security concepts and controls in response to changing security and safety requirements. The development methods and system architectures to be explored should allow for the secure and safe integration of untrustworthy or unchangeable system parts, even if those parts might have weaknesses. The resulting system should detect and isolate software subsystem failures and enable a recovery in case one or multiple subsystems are compromised. We will investigate concepts like secure confinement, compartmentalization, diversification, and formal proofs of correct operation to deal with novel attack scenarios.

6.4 Efficient and Scalable Consensus

Consensus protocols provide a solution to many of the above challenges, especially in terms of dealing with the detection and recovery of subsystem failure. Recently there have been significant advances in improving the security and efficiency of distributed consensus by leveraging available hardware security mechanisms. For example, MinBFT [51] uses the trustworthy USIG service (see Section 6.2) to prevent replicas equivocating and thus reduces the total number of replicas required to tolerate f failures from n=3f+1 to n=2f+1. Similarly, FastBFT [34] uses a hardware-based TEE to significantly improve efficiency and scalability of consensus with equivalent security guarantees.

However, although hardware security mechanisms bring clear advantages, reliance on such mechanisms could also limit the scenarios in which the consensus protocol could be used. For example, if even one of the replicas does not have access to a suitable trustworthy component, neither MinBFT nor FastBFT can be used. Real-world autonomous systems could be made up of heterogeneous components (e.g. different types of autonomous vehicles in a platoon).

Therefore it is still an open research problem to develop efficient consensus mechanisms whilst minimizing assumptions about the underlying hardware. In this line of research we aim to develop new consensus protocols and techniques that are able to opportunistically make use of available hardware security features, but can also be used in cases where only a subset of replicas supports these features.

6.5 Appropriate Consideration of Assumptions

Undoubtedly, assumptions are instrumental for implementing any real-world system. However, at the same time experience shows that the violation of assumptions has been the cause for many disasters and security breaches. Often assumptions are deeply rooted in paradigms one uses (e.g. low-skew clock distribution in synchronous design) or existing results one is building on (e.g. correctness proofs of algorithms). Therefore, careful reflection of all assumptions and their coverage is crucial for obtaining high safety and security, even more so in the context of sustainability with unknown future challenges lying far ahead of the actual design phase.

Consequently, we consider it one of the formidable challenges for S3 to (a) generally avoid making assumptions wherever possible, and (b) identify, justify and carefully validate every single assumption that is made. This encompasses all the examples given in Section 5.12, like "independence" of subsystem failures, or "isolation" between processes. Their validation must definitely entail the identification, comprehension and modeling of the relevant unintended and non-functional behaviors of hardware and software, or, alternatively, some kind of (ideally formal) evidence.

6.6 Trustworthy and Reliable Hardware

Modern computer systems and the underlying hardware are becoming increasingly faster, efficient, interconnected and consequently more complex introducing the possibility of new bugs and security related vulnerabilities. The semiconductor industry employs a combination of different verification techniques to ensure the quality and security of System-on-Chip (SoC) designs during the development life cycle. However, a growing number of increasingly sophisticated attacks are starting to leverage *cross-layer bugs* by exploiting subtle interactions between hardware and software, as recently demonstrated through a series of real-world exploits with significant security impact that affected all major hardware vendors.

A deep dive into microarchitectural security from a hardware designer's perspective seems to be indispensable. Currently there exists a protection gap in practice that leaves large chip designs vulnerable to software-based attacks. In particular, existing verification approaches fail to detect specific classes of vulnerabilities, i.e., bugs that evade detection by current verification techniques while being exploitable from software. We have already observed such vulnerabilities in real-world SoCs. Patching these hardware bugs may not always be possible and can potentially result in a product recall.

Our recently conducted hardware security competition affirmed that current verification approaches are insufficient.[17] In the competition 54 independent teams of researchers competed world-wide over a period of twelve weeks to catch inserted security bugs in SoC RTL designs, and an in-depth systematic evaluation of state-of-the-art verification approaches. Our findings indicate that even combinations of techniques will miss high-impact bugs due to the large number of modules with complex interdependencies as well

[17] https://hack-dac18.trust-sysec.com/

as fundamental limitations of current detection approaches. Indeed we showed that we can craft a real-world software attack that exploits one of the RTL bugs that evaded detection. These results show clearly that there is an urgent need for novel and effective methods to analyze the security of hardware designs at the deepest level and to avoid hardware/firmware bugs that can be exploited (remotely) by software.

7 CONCLUSION

Achieving sustainable security & safety in real-world systems is a non-trivial challenge. It goes well beyond our current paradigms for building *secure* systems because it calls for consideration of safety aspects and anticipation of threats beyond the foreseeable threat horizon. It also goes beyond our current thinking about *safety* by broadening the scope to include deliberate faults induced by an adversary. Nevertheless, sustainable security & safety will become increasingly necessary as we become increasingly dependent on these (ICT) systems. Even if the full vision of sustainable security & safety is not fully achieved, any advances in this direction could have a significant impact on the design of future system. We have set out our vision for sustainable security & safety, identified the main challenges currently faced, and proposed a set of design principles towards overcoming these challenges and achieving this vision.

8 ACKNOWLEDGEMENTS

This work was supported by the Intel Research Institute for Collaborative Autonomous and Resilient Systems (ICRI-CARS). The authors thank Muhammad Shafique for his helpful suggestions on this manuscript.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security* 13 (2009).

[2] National Highway Traffic Safety Administration. 2006. Vehicle Survivability and Travel Mileage Schedules . (2006). https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/809952

[3] Ross Anderson. 2018. Making Security Sustainable. *Commun. ACM* 61, 3 (Feb. 2018), 24–26. https://doi.org/10.1145/3180485

[4] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (Jan 2004), 11–33. https://doi.org/10.1109/TDSC.2004.2

[5] A. N. Bessani, P. Sousa, M. Correia, N. F. Neves, and P. Verissimo. 2008. The Crutial Way of Critical Infrastructure Protection. *IEEE Security Privacy* 6, 6 (Nov 2008), 44–51. https://doi.org/10.1109/MSP.2008.158

[6] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *37th IEEE Symposium on Security and Privacy (S&P)*.

[7] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. 2017. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *arXiv preprint arXiv:1709.09917* (2017).

[8] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies*.

[10] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. 2006. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. 2006 (2006).

[11] E. Brickell, G. Graunke, and J.-P. Seifert. 2006. Mitigating cache/timing attacks in AES and RSA software implementations. In *RSA Conference 2006, session DEV-203*.

[12] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium (USENIX Sec)*.

[13] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation.

[14] ENISA. 2013. *Algorithms, Key Sizes and Parameters Report - 2013 recommendations.* Technical Report. www.enisa.europe.eu.

[15] Ilias Giechaskiel, Cas Cremers, and Kasper B. Rasmussen. 2016. On Bitcoin Security in the Presence of Broken Cryptographic Primitives. In *Computer Security – ESORICS 2016*. Springer International Publishing, 201–222.

[16] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated whitebox fuzz testing. In *Annual Network & Distributed System Security Symposium (NDSS)*.

[17] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1 (Jan. 2012).

[18] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*.

[19] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Cache Attack to Induce Hardware Faults from a Website. In *13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*.

[20] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine. 2007. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems. In *2007 IEEE Symposium on Security and Privacy (SP '07)*. 281–295.

[21] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *2013 IEEE Symposium on Security and Privacy*.

[22] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. SSA: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *2015 IEEE Symposium on Security and Privacy*.

[23] ISO/TC 22/SC 32. 2011. ISO26262: Road vehicles - Functional safety. (2011).

[24] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *41st Annual International Symposium on Computer Architecture (ISCA)*.

[25] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[26] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '96)*.

[27] B.W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (Oct. 1973), 613–615.

[28] Éireann Leverett, Richard Clayton, and Ross Anderson. 2017. Standardisation and Certification of the 'Internet of Things'. https://www.cl.cam.ac.uk/~rja14/Papers/weis2017.pdf. (2017).

[29] Eireann Leverett, Richard Clayton, and Ross Anderson. 2018. Standardisation and Certification of Safety, Security and Privacy in the 'Internet of Things'. https://publications.europa.eu/en/publication-detail/-/publication/80bb1618-16bb-11e8-9253-01aa75ed71a1/language-en. (2018).

[30] Antonio Lima, Francisco Rocha, Marcus Völp, and Paulo Esteves-Veríssimo. 2016. Towards Safe and Secure Autonomous and Cooperative Vehicle Ecosystems. In *2ndACM Workshop on Cyber-Physical Systems Security and Privacy (co-located with CCS)*. Vienna, Austria.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[32] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*.

[33] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. 2016. Scalable Byzantine Consensus via Hardware-assisted Secret Sharing. *arXiv:1612.04997 [cs]* (Dec. 2016). http://arxiv.org/abs/1612.04997 arXiv: 1612.04997.

[34] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. 2018. Scalable Byzantine Consensus via Hardware-assisted Secret Sha ring. *IEEE Trans. Computers* (2018). https://doi.org/10.1109/TC.2018.2860009 https://doi.org/10.1109/TC.2018.2860009; Technical report version: [33].

[35] Rowan McAllister, Yarin Gal, Alex Kendall, Mark Van Der Wilk, Amar Shah, Roberto Cipolla, and Adrian Weller. 2017. Concrete Problems for Autonomous Vehicle Safety: Advantages of Bayesian Deep Learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*.

[36] T Mikolajick, A Heinzig, J Trommer, T Baldauf, and W M Weber. 2017. The RFET—a reconfigurable nanowire transistor and its application to novel electronic circuits and systems. *Semiconductor Science and Technology* 32, 4 (2017), 043001. http://stacks.iop.org/0268-1242/32/i=4/a=043001

[37] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990).

[38] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. *CacheZoom: How SGX Amplifies The Power of Cache Attacks*. Technical Report. arXiv:1703.06986 [cs.CR]. https://arxiv.org/abs/1703.06986.

[39] Peter Oehlert. 2005. Violating Assumptions with Fuzzing. *IEEE Security and Privacy* 3, 2 (March 2005).

[40] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symposium on Security and Privacy (S&P)*.

[41] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[42] Mark Seaborn and Thomas Dullien. 2016. Exploiting the DRAM rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.de/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. (2016).

[43] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[44] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613. https://doi.org/10.1145/359168.359176

[45] Kevin Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *34th IEEE Symposium on Security and Privacy (Oakland 2013)*.

[46] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. 2010. Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery. *IEEE Transactions on Parallel and Distributed Systems* 21, 4 (April 2010), 452–465. https://doi.org/10.1109/TPDS.2009.83

[47] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[48] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Commodity Mobile Platforms. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[49] Paulo Verissimo, Miguel Correia, Nuno Ferreira Neves, and Paulo Sousa. 2009. Intrusion-Resilient Middleware Design and Validation. In *Information Assurance, Security and Privacy Services*. Handbooks in Information Systems, Vol. 4. Emerald Group Publishing Limited, 615–678. http://www.navigators.di.fc.ul.pt/archive/papers/annals-IntTol-compacto.pdf

[50] Paulo Verissimo, Nuno Ferreira Neves, and Miguel Correia. 2003. Intrusion-Tolerant Architectures: Concepts and Design. In *Architecting Dependable Systems*. LNCS, Vol. 2677. Springer-Verlag, 3–36. http://www.navigators.di.fc.ul.pt/docs/abstracts/archit-03.html Extended version in http://hdl.handle.net/10455/2954.

[51] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (Jan. 2013), 16–30. https://doi.org/10.1109/TC.2011.221

[52] Marcus Völp, Francisco Rocha, Jeremie Decouchant, Jiangshan Yu, and Paulo Esteves-Verissimo. 2017. Permanent Reencryption: How to Survive Generations of Cryptanalysts to Come. In *Security Protocols XXV*, Frank Stajano, Jonathan Anderson, Bruce Christianson, and Vashek Matyáš (Eds.). Springer International Publishing, Cham, 232–237.

[53] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*.

PRELIMINARY LIST OF CHALLENGES

The following is a summarized list of the challenges presented in Section 3.

A Subsystem Failures

A.1 Hardware failure.

A.1.(1) How can we protect spares from environmental and adversarial influences such that they remain available when they are required?

A.1.(2) How can we assert the sustainability of emerging material circuits, without at the same time giving adversaries the tools to stress and ultimately break these circuits?

A.1.(3) How can we protect confidential information in subsystems or, if this is not possible over extended periods of time, how can we ensure confidential information is securely transferred and protected in its new location without residuals in the source subsystem revealing this information?

A.1.(4) How can we prevent one compromised hardware subsystem from compromising the integrity of another subsystem?

A.1.(5) How can we prevent adversaries from exploiting/triggering safety/security functionality of excluded components?

A.1.(6) How can we model erroneous behavior of hardware components in the presence of external disturbances?

A.1.(7) How can we construct inexpensive, fine grain isolation domains to confine such errors?

A.2 Software failure.

A.2.(1) How to design systems that can detect and isolate software subsystem failures?

A.2.(2) How to transfer software attack mitigation strategies between domains (e.g., PC to embedded)?

A.3 Subsystem compromise.

A.3.(1) How to recover a system when *multiple* subsystems are compromised?

A.3.(2) How to detect subsystem compromised by a stealthy adversary?

A.3.(3) How to react to the detection of a (potentially) compromised subsystem?

A.3.(4) How to prevent the leakage of sensitive information from a compromised subsystem?

A.3.(5) How to securely re-provision a subsystem after all its secrets have been leaked?

A.4 Specification failure.

A.4.(1) How to design subsystems that may fail at the implementation, but not at the specification level (and at what costs)?

A.4.(2) If specification faults are inevitable, how to design systems in which subsystems can follow different specifications whilst providing the same functionality, in order to benefit from diversity specifications and assumptions?

A.4.(3) How to recover when one of the essential systems has been broken due to a specification error (e.g., how to recover from a compromised cryptographic subsystem)?

B Requirement Changes

B.1 Regulatory changes.

B.1.(1) How to retroactively change the designed security, privacy, and/or safety guarantees of a system?

B.1.(2) How to prove that an already-deployed system complies with new regulations?

B.2 User expectation changes.

B.2.(1) How can a system be extended and adapted to meet new expectations after deployment?

B.2.(2) How to demonstrate to users and other stakeholders that an already-deployed system meets their new expectations?

B.3 Design lifetime changes.

B.3.(1) How to determine whether a deployed system will retain its safety and security guarantees for an even longer lifetime?

B.3.(2) How to further extend the safety and security guarantees of a deployed system?

C Environmental changes

C.1 New threat vectors.

C.1.(1) How to tolerate failure of subsystems due to unforeseeable threats?

C.1.(2) How to avoid single points of failure that could be susceptible to unforeseen threats?

C.1.(3) How to improve the modeling of couplings and dependencies between subsystems such that the space of "unforeseeable" threats can be minimized?

C.2 Unilateral system/service changes.

C.2.(1) How to design systems such that any third-party services on which they depend can be safely and securely changed?

C.2.(2) How can a system handle unilateral changes of (external) systems or services?

C.3 Third-party system/service fails.

C.3.(1) How can a system handle the failure or unavailability of external services?

C.3.(2) How to design systems such that any third-party services on which they depend can be safely and securely changed *after they have already failed*?

C.4 Maintenance resource becomes unavailable.

C.4.(1) How to identify all maintenance resources required by a system?

C.4.(2) How to maximize the *maintenance lifetime* of a system whilst minimizing cost?

C.4.(3) How to continue maintaining a system when a required resource becomes completely unavailable?

D Maintainer Changes

D.1 Implementing a change of maintainer.

D.1.(1) How to ensure that a system remains secure and safe even under a new maintainer?

D.1.(2) How to securely inform the system that a change of maintainer has taken place?

D.2 System becomes unmaintained.

D.2.(1) How can a system decide that it is no longer maintained?

D.2.(2) How should an unmaintained system behave?

E Ownership Changes

E.1 Cooperative ownership changes.

E.1.(1) How to securely inform the system about the change in ownership, without opening a potential attack vector?

E.1.(2) How to erase sensitive data during transfer of ownership, without allowing the previous owner to later erase usage/data of the new owner?

E.2 Non-cooperative ownership change.

E.2.(1) How to automatically detect non-cooperative ownership change?

E.2.(2) How to erase sensitive data after loss of ownership, without allowing the previous owner to erase usage/data of the new owner?