Sustainable Security & Safety: Challenges and Opportunities

Andrew Paverd
Microsoft Research Cambridge
andrew.paverd@ieee.org

Marcus Völp[1]
University of Luxembourg
marcus.voelp@uni.lu

Ferdinand Brasser[1]
TU Darmstadt, Germany
ferdinand.brasser@trust.tu-darmstadt.de

Matthias Schunter[1]
Intel Labs, Germany
matthias.schunter@intel.com

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

Ahmad-Reza Sadeghi[1]
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.

2012 ACM Subject Classification Security and privacy → Systems security; Software and its engineering → Software reliability

---

[1] corresponding author

4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS 2019).
Editors: Mikael Asplund and Michael Paulitsch; Article No. 4; pp. 4:1–4:13
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases sustainability, security, safety

Digital Object Identifier 10.4230/OASIcs.CERTS.2019.4

Related Version A full version of the paper is available at [3], `http://www.icri-cars.org/wp-content/uploads/2019/01/s3-vision.pdf`

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.

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,[2] 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[3], as shown in Figure 1. Worse, rarely any networked ICT system stays secure over such a lifetime, even if actively maintained.

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 [7]. For the currently envisioned limited lifetime, the (functional) safety community has developed techniques to prevent harm despite accidental faults [10]. 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.

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

---

[2] `http://www.dailymail.co.uk/news/article-2614323/Americas-feared-nuclear-missile-facilities-controlled-computers-1960s-floppy-disks.html`

[3] `https://www.aarp.org/money/budgeting-saving/info-05-2009/cars_that_last.html`

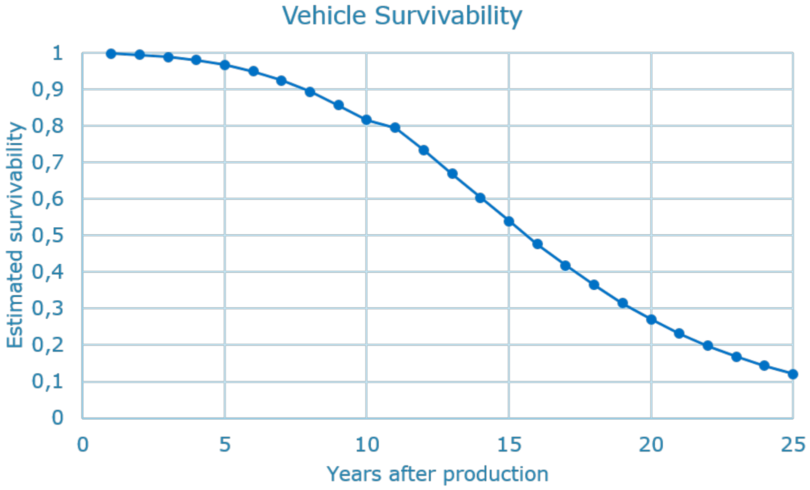

Figure 1 Survival probability of passenger cars by age [2] (based on 1977-2003 NVPP data)

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* [11] is a prototypical example of why established solutions from the dependability field cannot be directly used to address security hazards, and vice-versa. This 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. Subsystem failures may be hardware failures, software failures, subsystem compromises or specification failures. The open issues in these areas include:

Hardware failures: How can we protect spares (set our to take over in case of successful attacks or persistent faults) from environmental and adversarial influences, such that they remain available when they are required? How can we assert the sustainability of emerging material circuits, without at the same time giving adversaries the tools to stress and ultimatley break these circuits? How can we protect confidential information in subsystems? How can we prevent one compromised hardware subsystem from compromising the integrity of others? How can we prevent adversaries from exploiting safety/security functionality of excluded components? How can we model erroneous hardware behavior? And, how can we construct inexpensive, fine grain isolation domains to confine such errors?

Software failures: How to design systems that can detect and isolate software subsystem failures? How to transfer software attack mitigation strategies between domains (e.g., PC to embedded)?

Subsystem compromise: How to recover a system when *multiple* subsystems are compromised? How to detect subsystem compromised by a stealthy adversary? How to react to the detection of a (potentially) compromised subsystem? How to prevent the leakage of sensitive information from a compromised subsystem? How to securely re-provision a subsystem after all secrets have been leaked?

Specification failures: How to design subsystems that may fail at the implementation, but not at the specification level (and at what costs)? 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? 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)?

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. Requirement changes may be due to:

Regulatory changes: How to retroactively change the designed security, privacy, and/or safety guarantees of a system? How to prove that an already-deployed system complies with new regulations?

User expectation changes: How can a system be extended and adapted to meet new expectations after deployment? How to demonstrate to users and other stakeholders that an already-deployed system meets their new expectations?

Design lifetime changes: How to determine whether a deployed system will retain its safety and security guarantees for an even longer lifetime? How to further extend the safety and security guarantees of a deployed system?

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.

New threat vectors: How to tolerate failure of subsystems due to unforeseeable threats? How to avoid single points of failure that could be susceptible to unforeseen threats? How to improve the modeling of couplings and dependencies between subsystems such that the space of "unforeseeable" threats can be minimized?

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

Third-party system/service fails: 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*? How can a system handle the failure or unavailability of external services?

Maintenance resource becomes unavailable: How to identify all maintenance resources required by a system? How to maximize the *maintenance lifetime* of a system whilst minimizing cost? How to continue maintaining a system when a required resource becomes completely unavailable? How to ensure that a system remains secure and safe even under a new maintainer?

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.[4] 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. Maintenance change requires answering:

Implementing a change of maintainer: How to securely inform the system that a change of maintainer has taken place?

System becomes unmaintained: How can a system decide that it is no longer maintained? How should an unmaintained system behave?

---

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

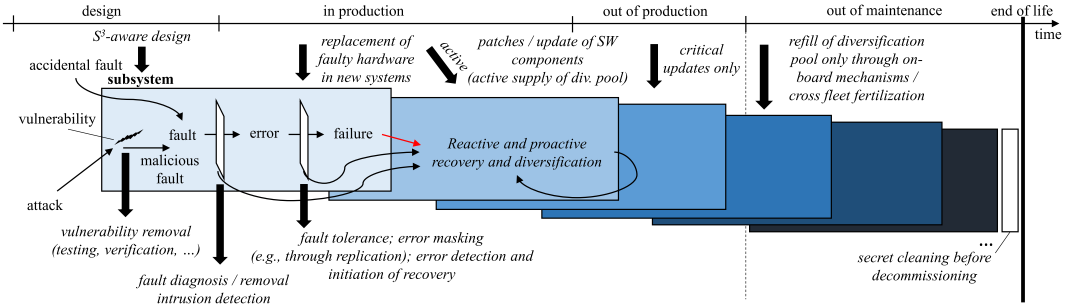

Figure 2 Means to attain dependability and security.

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). Ownership may be:

Cooperative: How to securely inform the system about the change in ownership, without opening a potential attack vector? How to erase sensitive data during transfer of ownership, without allowing the previous owner to later erase usage/data of the new owner?

Non-cooperative: How to automatically detect non-cooperative ownership change? 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 [11], 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.

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 [21, 20]. 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 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 [16], 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 [13, 14, 8, 9]. 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,[5] 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 the behaviour 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[6]) 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 (e.g., by mimicking

---

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

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

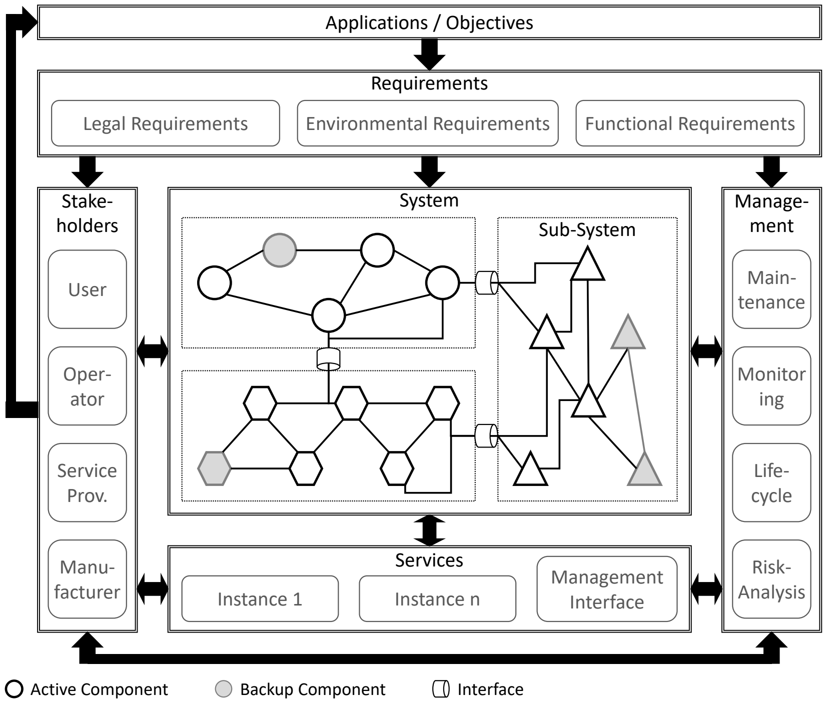

Figure 3 Abstract view of a sustainable system and its surroundings.

gene crosscopying in patch generation for defense). 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.[7]

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.[8]

---

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

[8] If an individual subsystem cannot provide the required fault tolerance level, the overall system requires

Sustainable systems usually do not operate in isolation. 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 it.

Based on the challenges discussed in the preceding section, we can already deduce certain architectural requirements and design principles for achieving S3:

- **Well-defined Components and Isolation**, limiting interaction to well-defined and constrained interfaces, to ensure complexity can be handled and to confine faults to causing components.
- **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. Anticipating this possibility, it is clear that no single subsystem can be responsible for the safety and security of the system as a whole.
- **Multiple Lines of Defense** Individual defenses can be overcome by adversaries, hence, relying on a single defense mechanism represents a single point of failure. Multiple lines of defense are needed to protect the system. Control-flow integrity (CFI) [1], for instance, has been attacked by full-function reuse attacks [6, 15] while randomization approaches suffer from information leakage [17]. Combining both can increase the security, e.g., by relying on randomization to mitigate the above attack to CFI.
- **Long-term Secrets and Confidentiality** Ensuring the confidentiality of data over long periods of time is very challenging since data, once leaked, cannot become confidential again. To prevent the leakage of secret information their use should be minimized. Additionally, 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.

  Let us give an example, detailled in [22]. Combining secret sharing, permanently re-encrypted data silos, function shipping into trusted execution environments, and sanitizing before revealing, it is possible to give access, even to bulk data, when it is required, but only in the form necessary. Rarely, applications require access to raw data. Instead, most just depend on aggregate, derived information which reveals much less of the secrets used. Revealing the classification of a deep neural network for a picture, while protecting the weights [19] is an example of such a secret protection scheme.
- **Robust Input Handling** (Sub)systems should not make assumptions with respect to the inputs it expects. This holds true for external as well as internal inputs. A robust system should incorporate: (a) it should cope well with noisy and uncertain real-world inputs, and (b) express its own knowledge and uncertainty of a proposed output despite unforeseen input [12].

---

redundancy with regard to that subsystem.

- **Contain Subsystem Failure** An immediate conclusion from the requirement to tolerate arbitrary failures 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.
- **Replicate to Tolerate Subsystem Failure** 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 to mask failure of individual replicas behind a majority of healthy replicas operating in consensus.
- **Diversify Nodes and Components** Replication is not sufficient to evade attacks [5]; diversification is needed to avoid common mode faults and cancel adversary knowledge [18].
- **Adaption and reconfiguration** is needed to stay ahead of adversaries, but also to adjust to environmental changes.
- **Minimize Assumptions** Attackers may find new ways of violating assumptions to defeat safety or security measures that rely on them. Minimizing assumptions to those substantially required for the implementation, mitigates this threat.
- **Simplicity and Verifiability** While due to performance demands it is out of reach to keep the whole system 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 threats.

6 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.

References

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

2 National Highway Traffic Safety Administration. Vehicle Survivability and Travel Mileage Schedules , 2006. URL: `https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/809952`.

3 Ferdinand Brasser Matthias Schunter N. Asokan Ahmad-Reza Sadeghi Paulo Esteves Verissimo Andreas Steininger Andrew Paverd, Marcus Völp and Thorsten Holz. Sustainable security & safety: Challenges and opportunities. long version avail. at: http://www.icri-cars.org/wp-content/uploads/2019/01/s3-vision.pdf, May 2019.

4 Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004. `doi:10.1109/TDSC.2004.2`.

5 Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreia Neves, and Paulo Verissimo. The crutial way of critical infrastructure protection. *IEEE Security Privacy*, 6(6):44–51, Nov 2008. `doi:10.1109/MSP.2008.158`.

6 Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium*, USENIX Sec, 2015.

7 ENISA. Algorithms, key sizes and parameters report - 2013 recommendations. Technical report, www.enisa.europe.eu, Oct. 2013.

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

9 Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1), January 2012.

10 ISO Technical Committee 22/SC 32. *ISO26262: Road vehicles - Functional safety*, 2018.

11 Antonio Lima, Francisco Rocha, Marcus Völp, and Paulo Esteves-Veríssimo. 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, October 2016.

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

13 Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12), December 1990.

14 Peter Oehlert. Violating Assumptions with Fuzzing. *IEEE S&P*, 3(2), March 2005.

15 Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symp. on Security and Privacy*, 2015.

16 Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979. URL: `http://doi.acm.org/10.1145/359168.359176`, `doi:10.1145/359168.359176`.

17 Kevin Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. 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)*, 2013.

18 Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreia Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, April 2010. `doi:10.1109/TPDS.2009.83`.

19 Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure DNN inference. *CoRR*, abs/1810.00602, 2018. URL: `http://arxiv.org/abs/1810.00602`, `arXiv:1810.00602`.

20 Paulo Verissimo, Miguel Correia, Nuno Ferreira Neves, and Paulo Sousa. Intrusion-resilient middleware design and validation. In *Information Assurance, Security and Privacy Services*, volume 4 of *Handbooks in Information Systems*, pages 615–678. Emerald Group Publishing Limited, May 2009. URL: `http://www.navigators.di.fc.ul.pt/archive/papers/annals-IntTol-compacto.pdf`.

21 Paulo Verissimo, Nuno Ferreira Neves, and Miguel Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*, pages 3–36. Springer-Verlag, June 2003. Extended version in http://hdl.handle.net/10455/2954. URL: `http://www.navigators.di.fc.ul.pt/docs/abstracts/archit-03.html`.

22 Marcus Völp, Francisco Rocha, Jeremie Decouchant, Jiangshan Yu, and Paulo Esteves-Verissimo. Permanent reencryption: How to survive generations of cryptanalysts to come. In Frank Stajano, Jonathan Anderson, Bruce Christianson, and Vashek Matyáš, editors, *Security Protocols XXV*, pages 232–237, Cham, 2017. Springer International Publishing.