Fundamental Trustworthiness Principles

Peter G. Neumann, SRI International, Menlo Park CA

Working draft, March 6, 2017

Abstract

Enormous benefits can result from basing requirements, architectures, implementations, and operational practices on well-defined and well-understood generally accepted principles. Furthermore, any set of principles is by itself clearly incomplete. However, considerable experience, understanding, and foresight are needed to use such principles productively.

In this document, we itemize, review, and interpret various design and development principles that – if properly understood and applied – can advance predictable composability of components, total-system trustworthiness, high assurance, and other attributes of computer systems and networks. We consider the relative applicability of those principles, as well as some of the problems they may introduce. We also examine the pervasive way in which these design and development principles have inspired and motivated the prototype CHERI architecture.

1 Background

System trustworthiness is a measure of how worthy a system might be to trusted to satisfy whatever critical requirements are desired, often relating to security, reliability, guarantees of real-time performance and resource availability, survivability, and more – all in spite of a wide range of adversities (known and some unknown). Trustworthiness depends on hardware, software, communications media, power supplies, physical environments, and ultimately people in many capacities – requirements specifiers, designers, implementers, users, operators, maintenance personnel, administrators, and (unfortunately) the abilities of attackers to exploit weaknesses or limitations in all of the above.

We examine here the exent to which the ongoing CHERI (Capability Hardware Enhanced RISC Instructions) system hardware-software co-design effort has successfully applied those principles (either intentionally or serendipitously). Recent CHERI papers include [1, 2, 3].

The prototype CHERI Instruction-Set Architecture (ISA) is an extension of the MIPS-64 ISA that adds capability instructions and capability registers. The CHERI ISA prototype provides support for strongly-typed objects, rapid domain crossing, very-fine-grained access controls, and compartmentalization of software. The clean-slate approach provides remarkably strong security, including adaptations of FreeBSD and LLVM-based compilers that understand the capability instructions, and that effectively prevent many of the common programming errors – or otherwise help programmers avoid them. It provides a hybrid architecture that allows legacy object code and recompiled source code (even with potential malware) to coexist securely with high-security software, without adverse effects. The hardware and software are open-sourced. Extensive technology transfer is in progress, currently halfway through year 7 of what is now an 8-year DARPA project (originally scheduled for only four years).

2 Introduction to Principled Systems

Everything should be made as simple as possible – but no simpler.
Albert Einstein

A fundamental hypothesis motivating this analysis is that achieving assurably trustworthy systems requires much greater observance of certain underlying principles than is normally present. We assert that careful attention to such principles can greatly facilitate the following efforts.

- **Principled architectures.** Establishment of predictably composable open distributed-system network-oriented architectures needs to rely on sensible total-system architectures if the systems are to be capable of fulfilling critical requirements (e.g., for security, reliability, survivability, and performance), while being readily adaptable to widely differing applications, different hardware, heterogeneous software providers, and changing technologies. Here, the term *architecture* here generally implies both the structure of systems and networks. and the design of their functional interfaces and interconnections – at various layers of abstraction. (This is distinct from the so-called hardware instruction-set architecture, which is concerned primarily with precisely defining the hardware interface to the software.)

- **Principled system development.** The entire development process needs to be sensibly structured and managed. This may include the development of detailed specifications, principled implementations that follow good coding practices, up-front concerns for trustworthiness, and assurance of that trustworthiness for composable interoperable components, with predictable behavior when those components are composed.

- **Principled assurance.** Any meaningful measure of trustworthiness requires serious attention to assurance that the conceptual requirements, abstract specifications, detailed implementations, and operational practices have some realistic justifiable expectation of satisfying the desired mission needs. Thus, attainment of assuredly trustworthy systems and networks that are capable of addressing all relevant critical requirements requires assurance methodologies that are themselves highly principled and that take advantage of the design, development, and operational principles.

The benefits of disciplined and principled system development cannot be overestimated, especially in the early stages of the development cycle. Principled design and software development can stave off many problems later on in implementation, maintenance, and operation. Huge potential cost savings can result from diligently observing and maintaining relevant principles throughout the design and development cycles. However, the primary concept involved is that of disciplined development; there are many methodologies that provide some kind of discipline, and all of those can be useful in some cases. Furthermore, no system can be called "trustworthy" in the absence of meaningful assurance evaluations.

Many of the principles discussed here are fairly well known in concept, and reasonably well understood by system cognoscenti. However, their relevance is often not generally appreciated by people with little development or operational experience. Not wishing to preach to the choir, we do not dwell on elaborating the principles themselves, which have been extensively covered elsewhere (as cited below). Instead, we concentrate on the importance and applicability of these principles in the development of systems with critical requirements – and especially secure systems and networks. The clear implication is that disciplined understanding and observance of the most

effective of these principles can have enormous benefits to developers and system administrators, and also can aid user communities. However, we also explore various potential conflicts within and among these principles, and emphasize that those conflicts must be thoroughly understood and respected. The challenges in developing trustworthy systems development are intrinsically complicated, especially when attempting to meet life-critical or other stringent requirements. For example, it is important to find ways to manage that complexity, rather than mistakenly believing that intrinsic complexity is avoidable by pretending to practice "simplicity".

3 Trustworthiness Principles

> *Willpower is always more efficient than mechanical enforcement, when it works. But there is always a size of system beyond which willpower will be inadequate.*
> Butler Lampson

Developing and operating complex systems and networks with critical requirements demands a different kind of thinking from that used in operating-system design and routine programming. We consider here various sets of principles, their applicability, and their limitations. We begin with the historically significant Saltzer-Schroeder-Kaashoek principles, followed by several other sets of principles and structural developmental approaches.

Of particular interest here are compositions of different elements (e.g., requirements, specifications, implementations, and analyses), and the assurance that can be attributed to systems with requirements for trustworthiness. Critical problems relate to composability (preservation of existing properties) and compositionality (analysis of emergent properties arising from the compositions) of constituent components, some of which can be extremely difficult to predict. Examples of compositionality include total-system safety and security, which cannot be evaluated component-wise. These properties may be extremely difficult to predict, although failures that result may be evident (e.g., failure to work at all), or subtle, surprising, and very difficult to detect (e.g., [4] which is still relevant today – even if outdated), A recent example of surprising extensive failures of compositionality involved the client and server sides of essentially all popular implementations of the heavily used and very critical cryptographically based TLS 1.2 protocol [5].

3.1 Saltzer-Schroeder-Kaashoek Security Principles, 1975 and 2009

The ten basic security principles formulated by Saltzer and Schroeder [6] in 1975 are all still relevant today, in a wide range of circumstances. An eleventh principle on minimizing what must be trusted appears in the 2009 book by Saltzer and Kaashoek [7]. (I had included a similar principle on minimizing what has to be trusted to achieve survivable systems in my 2000 ARL report [8] and again in my 2004 DARPA report on problems and approaches related to trustworthy compositions [4].) In essence, these principles are summarized here, along with CHERI-relevant explanations. In addition, the *principle of intentional use* has been added as a twelfth basic principle, at the suggestion of Robert Watson. It is an excellent partner for the principle of least privilege, as it adds considerably more refinement to the indirect use of privileges on behalf of other processes (or users).

1. **Economy of mechanism:** Seek design simplicity (wherever and to whatever extent it is effective). CHERI's high-assurance hardware relies on just a handful of capability instruc-

tions, and compilers that constructively utilize those instructions, out of which extremely trustworthy systems and applications can be constructed.

2. **Fail-safe defaults:** Deny accesses unless explicitly authorized (rather than permitting accesses unless explicitly denied). CHERI process initializations can minimize what privileges are available. Further CHERI has no Hydra-like amplification of capability privileges, providing strict monotonicity of privileges.

3. **Complete mediation:** Check every access, without exception. CHERI capabilities are unforgeable, and the capability mechanism is nonbypassable. Any attempts to modify a capability results in something that is no longer a capability.

4. **Open design:** Do not assume that design secrecy will enhance security. The CHERI Instruction Set Architectures (for both the 256-bit research capabilities and the 128-bit implementable version) is open-sourced.

5. **Separation of privileges:** Use separate privileges or even multiparty authorization (e.g., two keys) to reduce misplaced trust. The CHERI ISA supports typed objects that enforce strong typing, each type having its own separately defined privileges that are relevant to the particular type.

6. **Least privilege:** Allocate minimal (separate) privileges according to need-to-know, need-to-modify, need-to-delete, need-to-use, and so on of the typed objects. The existence of overly powerful mechanisms such as *superuser* is inherently dangerous. The CHERI hardware capability mechanism in which each capability has its own minimal set of privileges and the hardware-software support for fine-grained compartmentalization further enhances the least-privilege principle.

7. **Least common mechanism:** Minimize the amount of mechanism common to more than one user and depended on by all users. Avoid sharing of trusted multipurpose mechanisms, including executables and data – in particular, minimizing the need for and use of overly powerful mechanisms such as *superuser* and nonlocally shared buffers. As one example of the flaunting of this principle, exhaustion of shared resources provides a huge source of covert storage channels, whereas the natural sharing of real calendar-clock time provides a source of covert timing channels. CHERI's strongly typed higher-layer objects allow each type to have its own rules and privileges; when reusing any common code, the capability mechanism and the compartmentalization provide suitable isolation to prevent harmful effects. On the other hand, at present covert channels are not considered in the CHERI ISA, and must be dealt with in implementation.

8. **Psychological acceptability:** Strive for ease of use and operation – for example, with easily understandable and forgiving interfaces. To a considerable extent, issues related to security benefit from being largely invisible to casual users. Much of the CHERI capability mechanism can be hidden by operating systems and compilers. Optimizing performance is likely to be not necessary for users other than application developers.

9. **Work factor:** Make cost-to-protect commensurate with threats and expected risks. This is a problem in many conventional systems, in which attackers need to find only a few exploitable weak links, whereas system developers and administrators need to ensure that

there are very few exploitable weak links that cannot be isolated. The work factor is often mistakenly applied to cryptography, when exploitable system flaws can totally undermine the belief that the cryptography is very strong. We believe that formal analysis of the CHERI ISAs can greatly increase the assurance associated with the trustworthiness of the hardware, and that the resulting software will be significantly less vulnerable to hardware-based attacks.

10. **Recording of compromises:** Provide nonbypassable tamperproof trails of evidence. This is a huge problem in proprietary paperless all-electronic voting machines and election support systems. CHERI can provide the nonbypassability and tamperproof trails of evidence through compartmentalization and fine-grained least privilege. This is a requirement that is made much easier on CHERI-based systems, because of the compartmentalization, encapsulation, control-flow integrity, and other trustworthiness attributes that can enable forensics-worthy audit trails. Thus, this principle is not one that guided the development, but rather emerged naturally as a by-product of the system itself.

11. **Minimization of what has to be trustworthy:** Poorly designed systems may have a sense of structural abstraction, but typically do not provide encapsulation (discussed in the next subsection) within each module – which often leads to vulnerabilities. In such systems, higher-layer abstractions can often compromise lower layers. CHERI's ISA and indeed its overall hardware-software system seriously pursue this principle throughout, with encapsulated abstractions that can be enforced with fine-grained least-privilege capability architecture in hardware, and fine-grained compartmentalization within processes in hardware and software – providing an elegant way to respect this principle. The following added principle is also highly beneficial.

12. **Intentional use.** Whenever multiple rights are available to a program, the selection of rights used to authorize work on behalf of the program must be explicit, irrespective of the specific layer of software abstraction. The intent of this principle is to avoid the accidental or unintended exercise of rights that could lead to a violation of the intended policy. It counters what is classically known as 'confused deputy' problems, in which a program can unintentionally exercise a privilege that it holds legitimately, but on behalf of another program that does not (and should not) have the ability to exercise that privilege. This principle (implicit in many capability systems) has been applied throughout the CHERI design, including architectural privileges (e.g., the requirement to explicitly identify capability registers used for load or store) and the sealed capability mechanism that can be used to support the CHERI object-capability model. (An attempt to satisfy this principle was found implicitly in PSOS's propagation-limiting capabilities, and has also been applied to CheriBSD.)

With regard to the penultimate principle above (11), appropriate trustworthiness should be situated where it is most needed – suitable to overall system requirements, rather than required uniformly across widely distributed components (with potentially many weak links) or totally centralized (with creation of a single weak link and forgetting other vulnerabilities). Trustworthiness is expensive to implement and to ensure; as a consequence, significant benefits can result from architecturally minimizing the extent to which higher-layer mechanisms have to trustworthy, especially if they can depend on (and rely on) the trustworthiness of lower layers. This principle

can contribute notably to sound architectures. In combination with economy of mechanism, this suggests avoidance of bloatware and unfortunate dependence on less trustworthy components.

Remember that these are principles, not hard-and-fast rules. By no means should they be interpreted as ironclad, especially in light of some of their potential mutual contradictions that require development tradeoffs. (See Section 4.) The Saltzer-Schroeder principles grew directly out of the Multics experience (e.g., [9]), discussed further at the end of this section. Some of these principles have taken on almost mythic proportions among the security elite, and to some extent buzzword cult status among many fringe parties. Therefore, perhaps it is not necessary to explain each principle in detail – although there is considerable depth of discussion underlying each principle. Careful reading of the Saltzer-Schroeder paper [6] and the Saltzer-Kaashoek book [7] is recommended. Matt Bishop's security books [10, 11] are also useful in this regard, placing the principles in a more general context. In addition, Chapter 6 of Matt Curtin's book [12] on "developing trust" – by which he might really hope to be "developing trustworthiness" – provides some useful further discussion of these principles. Also, consider the discussion below on additional principles in Section 3.2.

There are two fundamental caveats regarding these principles. First, each principle by itself may be useful in some cases and not in others. The second is that when taken in combinations, groups of principles are not necessarily all reinforcing; indeed, they may actually be mutually in conflict. Consequently, any sensible development must consider appropriate use of each principle in the context of the overall effort. Examples of a principle being both good and bad – as well as examples of cross-principle interference – are scattered through the following discussion. Various caveats are considered in the penultimate section.

Table 1 summarizes the applicability of each of the Saltzer-Schroeder-Kaashoek principles to the goals of composability, trustworthiness, and assurance (particularly with respect to security, reliability, and survivability-relevant requirements). Although this table is somewhat generic, it is also specifically relevant to CHERI, in light of the CHERI-relevant enumeration of the principles above. An asterisk indicates that CHERI actually makes constructive use of this principle in the system design (consciously or unconsciously), and thereby enhances trustworthiness. An asterisk in parenthesis implies that this principle was not a driving force in the design, but became easier to satisfy in implementations – as a result of the principled system architecture.

In particular, complete mediation, separation of privileges, and allocation of least privilege are enormously helpful to composability and trustworthiness. Open design can contribute significantly to composability, when subjected to internal review and external criticism. However, there is considerable debate about the importance of open design with respect to trustworthiness, with some people still clinging tenaciously to the notion that security by obscurity is sensible – despite risks of many flaws being so obvious as to be easily detected externally, even without reverse engineering. Indeed, the recent emergence of very good decompilers for C and Java, along with the likelihood of similar reverse engineering tools for other languages, both suggest that such attacks are becoming steadily more practical. Overall, the assumption of design secrecy and the supposed unavailability of source code is often not a deterrent, especially with ever-increasing skills among black-box system analysts. However, there are of course cases in which security by obscurity is unavoidable – as in the hiding of private and secret cryptographic keys, even where the cryptographic algorithms and implementations are public.

Fundamental to trustworthiness is the extent to which systems and networks can avoid being compromised by malicious or accidental human behavior and by events such as hardware malfunctions and so-called acts of God. In [8] and elsewhere, we have considered **compromise**

Table 1: Relevance of Saltzer-Schroeder to CHERI Goals

Principle	**Composability**	**Trustworthiness**	**Assurance**
Economy of mechanism *	Beneficial within a sound architecture; requires proactive design effort	Vital aid to sound design; exceptions must be completely handled	Can simplify analysis
Fail-safe defaults *	Some help, but not fundamental	Simplifies design, use, operation	Can simplify analysis
Complete mediation *	Very beneficial with disjoint object types	Vital, but hard to achieve without hardware help	Can considerably simplify analysis Nonbypassability
Open design *	Design documentation is very beneficial among multiple developers	Secrecy of design is, a bad assumption; open design requires strong system security	Assurance is mostly irrelevant in badly designed systems; open design enables total-system analysis
Separation of privileges *	Very beneficial if hardware supported	Avoids many common flaws	Focuses analysis more precisely
Least privilege *	Very beneficial if hardware supported	Limits flaws; improves design and operation	Focuses analysis more precisely
Least common mechanism *	Beneficial absent natural polymorphism	Avoids some common flaws	Modularizes analysis
Psychological acceptability	Could help a little – if not subvertible	Affects mostly usability and operations	Ease of use can contribute
Work factor *	Relevant especially for crypto algorithms but not implementations; may not be composable	Misguided if system easily compromised from below, spoofed, bypassed, etc.	Gives false sense of security under non-algorithmic compromises
Compromise recording (*)	Not an impediment if distributed; real-time detection/response needs must be anticipated	After-the-fact, but useful, easy to attain in secure systems	Not primary contributor to analysis
Minimize what must be trustworthy *	Composability can be significantly improved	Can greatly increase trustworthiness	Formal analysis and flow control can detect flaws
Intentional use of rights *	Simplifies predictable composability	Enhances least privilege	Refines operational assurance

from outside, compromise from within, and **compromise from below,** with fairly intuitive meanings. For example, outsiders may penetrate a system, or create denials of service; insiders may be able to masquerade as other users or misuse existing privileges; operating systems may be compromised from below by utilizing hardware quirks, and applications may be compromised by manipulating operating systems. There are cases in theory where weak links can be avoided

(e.g., end-to-end encryption for integrity, and zero-knowledge protocols that can establish a shared key without any part of the protocol requiring secrecy), although in practice they may also be undermined by compromises from below.

From its beginning, the Multics development was strongly motivated by a set of principles – some simple ones were originally stated by Ted Glaser and Peter Neumann in the first section of the very first edition of the Multics Programmers' Manual in 1965. (See http://multicians.org.) Multics was also driven by disciplined development. For example, with almost no exceptions, coding effort was never begun until a written specification had been approved by the Multics advisory board; also with almost no exceptions, all of the code was written in a subset of PL/I just sufficient for the initial needs of Multics, for which the first compiler (early PL, or EPL) had been developed by Doug McIlroy and Bob Morris.

In addition to the Saltzer-Schroeder principles, further insights on principles and discipline relating to Multics can be found in a paper by Corbató, Saltzer, and Clingen [13] and in Corbató's Turing lecture [14].

3.2 Related Principles, 1969 and Later

Another view of principled system development was given by Neumann in 1969 [15], relating to what is often dismissed as merely "motherhood" – but which in reality is both very profound and difficult to observe in practice. The basic motherhood principles under consideration in that paper (alternatively, you might consider them just as desirable system attributes) included automatedness, availability, convenience, debuggability, documentedness, efficiency, evolvability, flexibility, forgivingness, generality, maintainability, modularity, monitorability, portability, reliability, simplicity, and uniformity. Some of those attributes indirectly affect security and trustworthiness, whereas others affect the acceptability, utility, and future life of the systems in question. Considerable discussion in [15] was also devoted to (1) the risks of local optimization and the need for a more global awareness of less obvious downstream costs of development (e.g., writing code for bad – or nonexistent – specifications, and having to debug really bad code), operation, and maintenance; and (2) the benefits of higher-level implementation languages (which prior to Multics were rarely used for the development of operating systems [14, 13]).

In later reports (e.g. [8]), Neumann considered some extensions of the Saltzer-Schroeder principles. Although most of those principles might seem more or less obvious, they are of course full of different interpretations and hidden issues. We summarize an extended set of principles here, particularly as they might be interpreted in the CHERI context.

13. **Sound architectures.** Recognizing that it is much better to avoid design errors early than to attempt to fix them later, the importance of architectures inherently capable of evolvable, maintainable, robust implementations is enormous – even in an open-source environment. The value of a well-thought-out architecture is considerable in open-source systems. The value in closed-source proprietary systems could also be significant, if it were thought through early on, although architectural foresight is often impeded by legacy compatibility requirements that tend to lock system evolution into inflexible architectures. Good interface design is as fundamental to good architectures as is their structure. Both the architectural structure and the architectural interfaces (particularly the visible interfaces, but also some of the internal interfaces that must be interoperable) benefit from careful early specification. Defense in depth and defense in breadth are both conceptually desirable, but only in the context of the preceding and following principles as they relate to total-system trustworthiness.

14. **Abstraction.** The primitives at any given logical or physical layer should be relevant to the functions and properties of the objects at that layer, and should mask lower-layer detail where possible. Ideally, the specification of a given abstraction should be in terms of objects meaningful at that layer, rather than requiring lower-layer (e.g., machine dependent) concepts. Abstractions at one layer can be related to the abstractions at other layers in a variety of ways, thus simplifying the abstractions at each layer rather than collapsing different abstractions into a more complex single layer. (Horizontal and vertical abstractions and six types of abstractions are discussed in Virgil Gligor's contributed appendix on Visibly Controllable Computing in [4]. This text is an elaboration of David Parnas's "uses" paper [16].)

15. **Modularity.** Modularity relates to the characteristic of system structures in which different entities (modules) can be relatively loosely coupled and combined to satisfy overall system requirements, whereby a module could be modified or replaced as long as the new version satisfies the given interface specification. In general, modularity is most effective when the modules reflect specific abstractions and provide encapsulation within each module. CHERI takes modularity seriously, and actually provides submodularization particularly in application software when a particular application module needs further separation.

16. **Encapsulation.** Details that are relevant to a particular abstraction should be local to that abstraction and subsequently isolated within the implementation of that abstraction and the lower layers on which the implementation depends. One example of encapsulation involves information hiding – for example, keeping internal state information hidden from the visible interfaces. Another example involves masking the idiosyncrasies of physical devices from higher-layer system interfaces – and of course from the user interfaces as well. Encapsulation includes but is not limited to *information hiding* (as in the early work of David Parnas), and also helps maintain integrity of the abstraction in question from manipulation from outside the modular abstraction. The CHERI hardware ISA supports encapsulation in several respects, including within typed objects, and also as a by-product of the Bluespec strongly typed language that has been used to specify our various prototype ISAs.

17. **Layered and compositional assurance.** Protection (and generally defensive design for security, reliability, and so on) should be distributed to where it is most needed, and should reflect the semantics of the objects being protected. Layering (e.g., Multics rings or Dijkstra's THE system) can be very effective without losing efficiency. Compositional separation (compartmentalization) among modules or even within a single application or modular abstraction can also be effective. Structured abstractions can greatly simplify analysis, although the compositions themselves must also be analyzed. With respect to the reality of implementations that transit entities of different trustworthiness, layers of protection are vastly preferable to flat concepts such as single sign-on (that is, where only a single authentication is required). With respect to psychological acceptability, single signon has enormous appeal; however, it can leave enormous security vulnerabilities as a result of compromise from outside, from within, or from below, in both distributed and layered environments. Thus, with respect to the apparent user simplicity provided by single signon, psychological acceptability conflicts with other principles, such as complete mediation, separation of privileges, and least common privilege. The hierarchically layered separation of the CHERI hardware and the various software layers as well as the horizontal separations provided by compartmentalization are fundamental to CHERI's trustworthiness.

18. **Constrained dependency.** Improperly guarded dependencies on less trustworthy entities should be avoided. However, it is possible in some cases to surmount the relative untrustworthiness of mechanisms on which certain functionality depends – as in the two-dozen types of trustworthiness-enhancing mechanisms enumerated in [4]. In essence, do not trust anything on which you must depend – unless you are seriously satisfied with demonstrations of its trustworthiness. This principle is a generalization of the Biba property [17], which deals more specifically with multilevel integrity.

19. **Object orientation.** The OO paradigm bundles together abstraction, encapsulation, modularity of state information, inheritance (subclasses inheriting the attributes of their parent classes – e.g., for functionality and for protection), and subtype polymorphism (subtype safety despite the possibility of application to objects of different types). This paradigm facilitates programming generality and software reusability, and if properly used can enhance software development. This is a contentious topic, in that most of the OO methodologies and languages are somewhat sloppy with respect to inheritance. (Jim Horning noted that the only object-oriented language he knows that takes inheritance of specifications seriously was the Digital Equipment Corporation ESL OWL/Trellis, which was a descendant of Barbara Liskov's CLU language.) CHERI supports the separations associated with typed objects, in both hardware and software.

20. **Separation of policy and mechanism.** Statements of policy should avoid inclusion of implementation-specific details. Furthermore, mechanisms should be policy-neutral where that is advantageous in achieving functional generality. However, this principle must never be used in the absence of understanding about the range of policies that might be usefully implemented. There is a temptation to avoid defining meaningful policies, deferring them until later in the development – and then discovering that the desired policies cannot be realized with the given mechanisms. This is a characteristic chicken-and-egg problem with abstraction. However, it is again fundamental to the CHERI total-system architecture.

21. **Separation of duties.** In relation to separation of privileges, separate classes of duties of users and computational entities should be identified, so that distinct system roles can be assigned accordingly. Distinct duties should be treated distinctly, as in system administrators, system programmers, and unprivileged users.

22. **Separation of roles.** Concerning separation of privileges, the roles recognized by protection mechanisms should correspond in some readily understandable way to the various duties. For example, a single all-powerful superuser role is intrinsically in violation of separation of duties, separation of roles, separation of privilege, and separation of domains. The separation of would-be superuser functions into separate roles as in Trusted Xenix is a good example of desirable separation. Once again (as with single signon, noted above), there is a conflict between principles: the monolithic superuser mechanism provides economy of mechanism, but violates other principles. In practice, all-powerful mechanisms are sometimes unavoidable, and sometimes even desirable despite the negative consequences (particularly if confined to a secure sub-environment). However, they should be avoided wherever possible.

23. **Separation of domains.** Concerning separation of privileges, domains should be able to enforce separate roles. For example, a single all-powerful superuser mechanism is inherently unwise, and is in conflict with the notion of separation of privileges. However, separation of

privileges is difficult to implement if there is inadequate separation of domains. Separation of domains can help enforce separation of privilege, but can also provide functional separation as in the Multics ring structure, a kernelized operating system with a carefully designed kernel, or a capability-based architecture.

24. **Sound authentication.** Authentication is a pervasive problem. Nonbypassable authentication should be applicable to users, processes, procedures, and in general to any active entity or object. Authentication relates to evidence that the identity of an entity is genuine, that procedure arguments are legitimate, that types are properly matched when strong typing is to be invoked, and other similar aspects.

25. **Sound authorization and access control.** Authorizations must be correctly and appropriately allocated, and nonsubvertible (although they are likely to assume that the identities of all entities and objects involved have been properly authenticated – see sound authentication, above). Crude all-or-nothing authorizations are often riskful (particularly with respect to insider misuse and programming flaws). In applications for which user-group-world authorizations are inadequate, access-control lists and role-based authorizations may be preferable. Finer-grained access controls may be desirable in some cases, such as capability-based addressing and field-based database protection. However, knowing who has access to what at any given time should be relatively easy to determine.

26. **Administrative controllability.** The facilities by which systems and networks are administered must be well designed, understandable, well documented, and sufficiently easy to use without inordinate risks. This both a driving principle of the CHERI architecture and a by-product of sensible use of the systems.

27. **Comprehensive accountability.** Well-designed and carefully implemented facilities are essential for comprehensive monitoring, auditing, interpretation, and automated response (as appropriate). Thus, this principle should an *a priori* concern, as serious security and privacy issues must be carefully used relating to the overall accountability processes and audit data. CHERI addresses this need through its provisioning of trustworthy hardware and operating systems, and its ability to provide high-integrity application compartmentalization.

Table 2 summarizes the utility of the extended-set principles with respect to the three goals of the CHERI program acronym, as in Table 1. Once again, an asterisk indicates that CHERI actually makes constructive use of this principle, and is thereby enhances trustworthiness. An asterisk in parenthesis implies that this principle was not a driving force in the design, but became easier to satisfy in implements – as a result of the principled system architecture.

For an extensive further elaboration of abstraction, modularity, dependence, and more, see Virgil Gligor's appendix (Visibly Controllable Computing) that he contributed to [4].

At this point in our analysis, it should be no surprise that all of these principles can contribute in varying ways to many aspects of total-system trustworthiness – safety, security, reliability, survivability, and other -ilities. Ultimately, all of these properties are emergent properties of the total system, and cannot be determined from the components. Furthermore, many of the principles and -ilities are interrelated. We cite just a few of the interdependencies that must be considered.

Table 2: Relevance of Extended-Set Trustworthiness Principles to CHERI Goals

Principle	**Composability**	**Trustworthiness**	**Assurance**
Sound system architecture *	Can considerably facilitate composability and compositionality	Can greatly increase trustworthiness	Can increase assurance of design and simplify implementation analysis
Abstraction *	Very beneficial with suitable independence	Very beneficial if composable	Simplifies analysis by decoupling it
Encapsulation *	Very beneficial if properly done, enhances integration	Very beneficial if composable; avoids certain types of bugs	Localizes analysis to abstractions and their interactions
Modularity *	Very beneficial if interfaces and specifications well defined	Very beneficial if well specified; overmodularization impairs performance	Simplifies analysis by decoupling it and if modules are well specified
Layered and compositional assurance *	Very beneficial, but may impair performance	Very beneficial if noncompromisible from above/within/below	Structures analysis according to layers and their interactions
Robust dependency *	Beneficial: can avoid compositional conflicts	Beneficial: can obviate design flaws based on misplaced trust	Robust architectural structure simplifies analysis
Object/type integrity *	Beneficial, but labor-intensive; can be inefficient	Can be beneficial, but complicates coding and debugging	Can simplify analysis of design, possibly implementation also
Separation of policy/mech. *	Beneficial, but both must compose	Increases flexibility and evolution	Simplifies analysis
Separation of duties *	Helpful indirectly as a precursor	Beneficial if well defined/enforced	Can simplify analysis if well defined
Separation of roles *	Beneficial if roles non-overlapping	Beneficial if properly enforced	Partitions analysis of design and operation
Separation of domains *	Can simplify composition and reduce side effects	Allows finer-grain enforcement and self-protection	Partitions analysis of implementation and operation
Sound authentication (*)	Helps if uniformly invoked	Huge security benefits, aids accountability	Can simplify analysis, improve assurance
Sound authorization (*)	Helps if uniformly invoked	Reduces misuse, aids accountability	Can simplify analysis, improve assurance
Administrative control (*)	Composability helps controllability	Good architecture helps controllability	Control enhances operational assurance
Comprehensive accountability (*)	Composability helps accountability	Beneficial for post-hoc analysis	Can provide feedback for improved assurance

For example, authorization is of limited use without authentication, *whenever identity is important.* Similarly, authentication may be of questionable use without authorization. In some cases, authorization requires fine-grained access controls. Least privilege requires some sort of separation of roles, duties, and domains. Separation of duties is difficult to achieve if there is no separation of roles. Separation of roles, duties, and domains each must rely on a supporting architecture.

The comprehensive accountability principle is particularly intricate, as it depends critically on many other principles being properly invoked. For example, accountability is inherently incomplete without authentication and authorization. In many cases, monitoring may be in conflict with privacy requirements and other social considerations [18], unless extremely stringent controls are enforceable. Furthermore, trustworthy forensic evidence requires trustworthy systems in the first place. Separation of duties and least privilege are particularly important here. All accountability procedures are subject to security attacks, and are typically prone to covert channels as well. Furthermore, the procedures themselves must be carefully monitored. Who monitors the monitors? *(Quis auditiet ipsos audites?)*

4 Caveats on Applying the Principles

> *For every complex problem, there is a simple solution. And it's always wrong.*
> H.L. Mencken

As we noted above, the principles referred to here may be in conflict with one another if each is applied independently; in certain cases, the principles are not composable. In general, each principle must be applied in the context of the overall development. Ideally, greater effort might be useful to reformulate the principles to make them more readily composable, or at least to make their potential tradeoffs or incompatibilities more explicit. However, this is probably counterproductive, because judicious use of principles is not a cookbook exercise.

There are also various potentially harmful considerations that must be considered – for example, over-use, under-use, or misapplication of these principles, and certain limitations inherent in the principles themselves. Merely paying lip-service to a principle is clearly a bad idea; principles must be sensibly applied to the extent that they are appropriate to the given purpose. Similarly, all of the criteria-based methodologies have many systemic limitations (e.g., [19, 20]); for example, formulaic application of evaluation criteria is always subject to incompleteness and misinterpretation of requirements, oversimplification in analysis, and sloppy evaluations. However, when carefully applied, such methodologies can be useful and add discipline to the development process. Thus, we stress here the importance of fully understanding the given requirements and of creating an overall architecture that is appropriate for realizing those requirements, before trying to conduct any assessments of compliance with principles or criteria. And then, the assessments must be taken for what they are worth – just one piece of the puzzle – rather than over-endowed as definitive results out of context. Overall, there is absolutely no substitute for human intelligence, experience, and foresight.

The Saltzer-Schroeder principle of design simplicity is one of the most popular and commonly cited. However, it can be extremely misleading when espoused (as it commonly is) in reference to systems with critical requirements for security, reliability, survivability, real-time performance, and high assurance – especially when all of these requirements are necessary within the same system environment. Simplicity is a very important concept in principle (in the small), but complexity is

often unavoidable in practice (in the large). For example, serious attempts to achieve fault-tolerant behavior often result in roughly doubling the size of the overall subsystem or even the entire system. As a result, the principle of simplicity should really be one of managing complexity rather than trying to eliminate it, particularly where complexity is in fact inherent in the combination of requirements. Keeping things simple is indeed a conceptually wonderful principle, but often not achievable in reality. Nevertheless, *unnecessary* complexity should of course be avoided. The back-side of the Einstein quote at the beginning of Section 2 is indeed both profound and relevant, yet often overlooked in the overzealous quest for perceived simplicity.

An extremely effective approach to dealing with intrinsic complexity is through a combination of the principles discussed here, particularly abstraction, modularity, encapsulation, and careful hierarchical separation that architecturally does not result in serious performance penalties, well conceived virtualized interfaces that greatly facilitate implementation evolution without requiring changes to the interfaces or that enable design evolution with minimal disruption, and far-sighted optimization. In particular, hierarchical abstraction can result in relative simplicity at the interfaces of each abstraction and each layer, in relative simplicity of the interconnections, and perhaps even relative simplicity in the implementation of each module. By keeping the components and their interconnections conceptually simple, it is possible to achieve conceptual simplicity of the overall system or networks of systems despite inherent complexity. Furthermore, simplicity can sometimes be achieved through design generality, recognizing that several seemingly different problems can be solved symmetrically at the same time, rather than creating different (and perhaps incompatible) solutions.

Note that such solutions might appear to be a violation of the principle of least common mechanism, but not when the common mechanism is fundamental – as in the use of a single uniform naming convention or the use of a uniform and nonbypassable capability-based addressing mode that transcends different subtypes of typed objects. In general, it is riskful to have multiple procedures managing the same data structure for the same purposes. However, it can be very beneficial to separate reading from writing – as in the case of one process that updates and another process that uses the data. It can also be beneficial to reuse the same code on different data structures, although strong typing is then important.

One further unfortunate common practice that should be considered as an anti-principle is known as *security by obscurity.* This involves the fallacious belief that if something is never revealed to the public, it is more likely to remain secure. There are notorious counter-examples, such as Matt Blaze's ability to render the Clipper Chip key-escrow process completely useless by disabling the Law-Enforcement Access Field (LEAF) without any access to the classified algorithms and classified production process. (I recall being told by an ex-NSA person: "Oh, yes, we knew about that vulnerability, but did not think anyone would find it.")

One of the primary goals of system developers should be to make system interfaces conceptually simple while masking complexity so that the complexities of the design process and the implementation itself can be hidden by the interfaces. This may in fact increase the complexity of the design process, the architecture, and the implementation. However, the resulting system complexity need be no greater than that required to satisfy the critical requirements such as those for security, reliability, and survivability. It is essential that tendencies toward bloatware be strongly resisted. (They seem to arise largely from the desire for bells and whistles – extra features – and fancy graphics, but also from a lack of enlightened management of program development.)

A networking example of the constructive use of highly principled hierarchical abstraction is given by the protocol layers of TCP/IP (e.g., [21]). An early total-system paper co-design is

given by the capability-based Provably Secure Operating System hardware-software paper design (PSOS) [22, 23, 24]), whose functionality at each of more than a dozen layers was specified formally in only a few pages each, with at least the bottom seven layers intended to be implemented in hardware. The underlying addressing is based on a capability mechanism (layer 0) that uniformly encompasses and protects objects of arbitrary types – including files, directories, processes, and other system- and user-defined types. The PSOS design is particularly noteworthy because a single capability-based operation at layer 12 (user processes) could be executed as a single machine instruction at layer 6 (system processes), with no iterative interpretation required unless there were missing pages or unlinked files that require operating system intervention (e.g., for dynamic linking of symbolic names, à la Multics). To many people, hierarchical layering instantly brings to mind inefficiency. However, the PSOS architecture is an example in which the hierarchical design could be implemented extremely efficiently – because of the power of the capability mechanism, strong typing, and abstraction, and its intended hardware implementation.

We note that formalism for its own sake is generally counterproductive. Formal methods are not likely to reduce the overall cost of software development, but can be helpful in decreasing the cost of software quality and assurance. They can be very effective in carefully chosen applications, such as evaluation of requirements, specifications, critical algorithms, and particularly critical code. Once again, we should be optimizing not just the cost of writing and debugging code, but rather optimizing more broadly over the life cycle.

There are many other common pitfalls that can result from the unprincipled use of principles. Blind acceptance of a set of principles without understanding their implications is clearly inappropriate. (Blind rejection of principles is also observed occasionally, particularly among people who establish firm requirements with no understanding of whether those requirements are realistically implementable – and among strong-willed developers with a serious lack of foresight.)

Lack of discipline is clearly inappropriate in design and development. For example, we have noted elsewhere [8, 25] that the open-source paradigm by itself is not likely to produce secure, reliable, survivable systems in the absence of considerable discipline throughout development, operation, and maintenance. However, with such discipline, there can be many benefits. (See also [26] on the many meanings of *open source,* as well as a Newcastle Dependable Interdisciplinary Research Collaboration (DIRC) final report [27] on dependability issues in open source, part of ongoing work.)

Any principle can typically be carried too far. For example, excessive abstraction can result in overmodularization, with enormous overhead resulting from intermodule communication and nonlocal control flow. On the other hand, conceptual abstraction through modularization that provides appropriate isolation and separation can sometimes be collapsed (e.g., for efficiency reasons) in the implementation – as long as the essential isolation and protection boundaries are not undermined. Thus, modularity should be considered where it is advantageous, but not merely for its own sake.

Application of each principle is typically somewhat context dependent, and in particular dependent on specific architectures. In general, principles should always be applied relative to the integrity of the architecture.

One of the severest risks in system development involves local optimization with respect to components or individual functions, rather than global optimization over the entire architecture, its implementation, and its operational characteristics. Radically different conclusions can be reached depending on whether or not you consider the long-term complexities and costs introduced by bad design, sloppy implementation, increased maintenance necessitated by hundreds of patches,

incompatibilities between upgrades, lack of interoperability among different components with or without upgrades, and general lack of foresight. Furthermore, unwise optimization (whether local or global) must not collapse abstraction boundaries that are essential for security or reliability – perhaps in the name of improved performance. As one example, real-time checks (such as bounds checks, type checking, and argument validation generally) should be kept close to the operations involved, for obvious reasons. As another example, the Risks Forum archives include several cases in which multiple alternative communication paths were specified, but were implemented in the same or parallel conduits – which were then all wiped out by a single backhoe!

Perhaps most insidious is the *a priori* lack of attention to critical requirements, such as any that might involve the motherhood attributes noted in [15] and listed above. Particularly in dealing with security, reliability, and survivability in the face of arbitrary adversities, there are few if any easy answers. But if those requirements are not dealt with from the beginning of a development, they can be extremely difficult to retrofit later. One particularly appealing survivability requirement would be that systems and networks should be able to reboot, reconfigure, and revalidate their soundness following arbitrary outages, without human intervention. That requirement has numerous architectural implications.

Once again, everything should be made as simple as possible, but no simpler. Careful adherence to principles that are deemed effective is likely to help achieve that goal.

5 Reviewing CHERI's Use of the Principles

Section 3.1 and Section 3.2 indicate that most of the principles enumerated here were instrumental (explicitly or even occasionally coincidentally) in the CHERI system hardware-software co-design and implementation – as summarized by the asterisks in the left-hand columns of Table 1 and Table 2.

Not surprisingly, the highly principled CHERI total-system architecture has actually succeeded in following most of these principles constructively – in the hardware ISAs, in low-layer software, and in the compilers. This principled approach is enabling considerable advances toward much greater trustworthiness. In particular, the CHERI hardware-software co-design has approached inherently complex problems architecturally, structuring the solutions to those problems as conceptually simple compositions of relatively simple components, with emphasis on the predictable behavior of the resulting systems and networks. We are also engaged in formal analyses of the critical hardware properties, which will enhance the assurance that the formal specifications of the hardware ISA will live up to our expectations. We hope that this carefully documented and highly principled effort will be an inspirational example to others.

In that the basic CHERI prototype hardware instruction-set architecture (256-bit capabilities on the extended MIPS64 ISA) is actually scalable downward (e.g., 128-bit capabilities, and even 64-bit capabilities on a 32-bit platform – without the memory-management unit) suggests a considerable range of applicability to a variety of applications. The high end would be very applicable to servers, cloud storage, rack computing, and powerful desktops; the medium version could be ideal for laptops and mobile devices; and the low end more suitable for devices and controllers for the Internet of Things. Thus, we can also envision a comparable range of trustworthy operating systems to match the power and trustworthiness of the capability hardware.

6 Conclusions

> *In theory, there is no difference between theory and practice. In practice, there is an enormous difference.* (Many variants of this concept are attributed to various people. This is a personal adaptation.)

What would be extremely desirable in our quest for trustworthy systems and networks is theory that is practical and practice that is sufficiently theoretical. Thoughtful and judiciously applied adherence to sensible principles appropriate for a particular development can greatly enhance the security, reliability, and overall survivability of the resulting systems and networks. These principles can also contribute greatly to operational interoperability, maintainability, operational flexibility, long-term evolvability, higher assurance, and many other desirable characteristics.

What are generally called "best practices" are often rather lowest-common-denominator techniques that have found their way into practice, rather than what might otherwise be the *best practices* that would be useful. (See [28, 29, 30]. On the other hand, citeNIST800-160 considers the engineering aspects that are relevant to reinforcing best practices and principles for enhancing cybersecurity.

Furthermore, the supposedly best practices can be mis-applied by very good programmers, and bad programming languages can still be used wisely. Unfortunately, spaghetti code is seemingly always on the menu, and engorged bloatware tends to win out over elegance. Overall, there are no easy answers.

Overall, having sensible system and network architectures is generally a good starting point – especially if they observe the principles noted here.

References

[1] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *Proceedings of the 41st International Symposium on Computer Architecture*, June 2014.

[2] R. N. Watson, P. G. Neumann, J. Woodruff, J. Anderson, D. Chisnall, B. Davis, B. Laurie, S. W. Moore, S. J. Murdoch, and M. Roe, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture, Version 1.14," tech. rep., SRI International and the University of Cambridge, September 2015.

[3] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Davé, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, (San Jose, California), IEEE Computer Society, May 2015.

[4] P. G. Neumann, "Principled assuredly trustworthy composable architectures," tech. rep., SRI Int'l, Menlo Park, CA, December 2004. http://www.csl.sri.com/neumann/chats4.html, .pdf.

[5] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A Messy State of the Union: Taming the Composite State Machines of TLS," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, (San Jose, California), IEEE Computer Society, May 2015. https://www.smacktls.com/smack.pdf.

[6] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, September 1975.

[7] J. H. Saltzer and F. Kaashoek, *Principles of Computer System Design*. Morgan Kaufmann, 2009. Chapters 1-6 only. Chapters 7-11 are online: http://ocw.mit.edu/Saltzer-Kaashoek.

[8] P. G. Neumann, "Practical architectures for survivable systems and networks," tech. rep., Final Report, Project 1688, SRI Int., Menlo Park, California, June 2000. http://www.csl.sri.com/neumann/survivability.html.

[9] E. I. Organick, *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, Massachusetts, 1972.

[10] M. Bishop, *Computer Security: Art and Science*. Addison-Wesley, Reading, Massachusetts, 2002.

[11] M. Bishop, *Introduction to Computer Security*. Addison-Wesley, Reading, Massachusetts, 2004.

[12] M. Curtin, *Developing Trust: Online Security and Privacy*. Apress, Berkeley, California, and Springer-Verlag, Berlin, 2002.

[13] F. J. Corbató, J. Saltzer, and C. T. Clingen, "Multics: The first seven years," in *Proceedings of the Spring Joint Computer Conference*, vol. 40, (Montvale, New Jersey), AFIPS Press, 1972.

[14] F. J. Corbató, "On building systems that will fail (1990 Turing Award Lecture, with a following interview by Karen Frenkel)," *Communications of the ACM*, vol. 34, pp. 72–90, September 1991.

[15] P. G. Neumann, "The role of motherhood in the pop art of system programming," in *Proceedings of the ACM Second Symposium on Operating Systems Principles, Princeton, New Jersey*, pp. 13–18, ACM, October 1969. http://www.multicians.org/pgn-motherhood.html.

[16] D. L. Parnas, "On a "buzzword": Hierarchical structure," in *Information Processing 74 (Proceedings of the IFIP Congress 1974)*, vol. Software, pp. 336–339, North-Holland, Amsterdam, 1974.

[17] K. Biba, "Integrity considerations for secure computer systems," Tech. Rep. MTR 3153, The Mitre Corporation, Bedford, Massachusetts, June 1975. Also available from USAF Electronic Systems Division, Bedford, Massachusetts, as ESD-TR-76-372, April 1977.

[18] D. Denning, P. G. Neumann, and D. B. Parker, "Social aspects of computer security," in *Proceedings of the 10th National Computer Security Conference*, September 1987.

[19] P. G. Neumann, "Rainbows and arrows: How the security criteria address computer misuse," in *Proceedings of the Thirteenth National Computer Security Conference*, (Washington, D.C.), pp. 414–422, NIST/NCSC, 1–4 October 1990.

[20] W. H. Ware, "A retrospective of the criteria movement," in *Proceedings of the Eighteenth National Information Systems Security Conference*, (Baltimore, Maryland), pp. 582–588, NIST/NCSC, 10–13 October 1995.

[21] C. Hunt, *TCP/IP Network Administration, 3rd Edition*. O'Reilly & Associates, Sebastopol, California, 2002.

[22] R. J. Feiertag and P. G. Neumann, "The foundations of a Provably Secure Operating System (PSOS)," in *Proceedings of the National Computer Conference*, pp. 329–334, AFIPS Press, 1979. http://www.csl.sri.com/neumann/psos.pdf.

[23] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson, "A Provably Secure Operating System: The system, its applications, and proofs," tech. rep., Computer Science Laboratory, SRI International, Menlo Park, California, May 1980. 2nd edition, Report CSL-116.

[24] P. G. Neumann and R. J. Feiertag, "PSOS revisited," in *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, (Las Vegas, Nevada), pp. 208–216, IEEE Computer Society, December 2003. http://www.acsac.org/ and http://www.csl.sri.com/neumann/psos03.pdf.

[25] P. G. Neumann, "Robust nonproprietary software," in *Proceedings of the 2000 Symposium on Security and Privacy*, (Oakland, CA), pp. 122–123, IEEE Computer Society, May 2000. http://www.csl.sri.com/neumann/ieee00.pdf.

[26] C. Gacek, T. Lawrie, and B. Arief, "The many meanings of open source," tech. rep., Department of Computing Science, University of Newcastle upon Tyne, Newcastle, England, August 2001. Technical Report CS-TR-737.

[27] C. Gacek and C. Jones, "Dependability issues in open source software," tech. rep., Department of Computing Science, Dependable Interdisciplinary Research Collaboration, University of Newcastle upon Tyne, Newcastle, England, 2001. Final report for PA5, part of ongoing related work.

[28] D. of Homeland Security, "Strategic principles for securing the internet of things," tech. rep., DHS, December 2016.

[29] BITAG, "Internet of things IoT security and privacy recommendations," tech. rep., BITAG Broadband Internet Technical Advisory Group, November 2016.

[30] C. on Enhancing National Cybersecurity, "Report on securing and growing the digital economy," tech. rep., NIST Publication, 1 December 2016, Gaithersburg MD, 1 December 2016.