Journal of Hardware and Systems Security
https://doi.org/10.1007/s41635-018-0047-0

SRASA: a Generalized Theoretical Framework for Security and Reliability Analysis in Computing Systems

Lake Bu[1] · **Jaya Dofe**[2] · **Qiaoyan Yu**[2] · **Michel A. Kinsy**[1]

Received: 15 February 2018 / Accepted: 27 August 2018

Abstract

Although there is a pressing need for highly secure and reliable computing systems, there is a glaring lack of formalism under which the properties of reliability and security can be jointly designed into these systems. This gap can primarily be attributed to the evolution of the two subfields. In the work, we introduce a unified generalized theoretical framework, called security and reliability aware state automaton (SRASA), to formally describe the specifications of a computer system that cover both security and reliability. SRASA is a 22-tuple finite state machine model that encompasses both physical and abstract states of the system, which may suffer from passive and active attacks. Three case studies illustrate the interpretation and application of the proposed SRASA theoretical framework. Our analysis and experimental results show that a non-physical attack may exploit unspecified or untested states to implement the malicious purpose, rather than introducing a new state to the system. To the best of our knowledge, this is the first attempt to bridge the current design specification gap between secure and reliable computing systems using a unified automaton approach. A general yet complete methodology that will encompass all aspects of system design, from the functional level specification to the gate level design at any system granularity, may not be feasible or it may be beyond the scope of a single work. Therefore, we aim in this work to (1) give an overview of the current landscape of reliability and security in systems design, (2) introduce a generalized framework to specify and reason about both reliability and security in the system design process, and finally (3) be general enough in the framework specification that it can be adopted or customized to more specific or concrete design instances.

Keywords Hardware · Reliability · Security · State machine · Testing · Evaluation

1 Introduction

The problem of secure and reliable computer system design is now a pressing issue. Cybersecurity incidents are happening at an astonishing frequency, scale, and sophistication. This problem will only exacerbate as we go deeper in the *Internet of Things* (IoT) era. IoT is growing three times faster than traditional computing platforms. Recent studies suggest that as many as 50 billion IoT devices will be installed by 2020 [30]. In fact, cybersecurity and reliability are now critical concerns in a wide range of embedded computing modules, communications systems, and connected devices. The traditional design approach of examining the security and reliability properties of a computing system in a two-step procedure is no longer viable because of their interplay. Currently, there is a glaring lack of formalism under which the properties of reliability and security can be jointly designed into computer systems.

In this work, we define "reliability" as the property of keeping the computing system in a known/functional/accepted set of states. Any deviation from this set of states due to

This research is partially supported by the NSF CAREER grant (No. CNS- 1652474) and NSF grant (No. CNS-1745808).

✉ Lake Bu
bulake@bu.edu

Jaya Dofe
jhs49@wildcats.unh.edu

Qiaoyan Yu
Qiaoyan.Yu@unh.edu

Michel A. Kinsy
mkinsy@bu.edu

[1] Department of Electrical and Computer Engineering, Adaptive and Secure Computing Systems Laboratory, Boston University, Boston, USA

[2] University of New Hampshire, Durham, NH, USA

Published online: 12 September 2018

random events or designer errors, i.e., a larger set of operations, will be deemed unreliable. In the case of "security," we define it to be a harmful attempt to either (1) create a deviation from the accepted set of states or (2) exploit the limitations of the functional set of states. This definition of security covers both active and passive attacks, direct state manipulation, and non-interfering information analysis.

From these definitions, one can already discern key similarities and differences between security and reliability. For example, non-interfering information analysis concerns in security do not extend to reliability. On the other hand, for state space deviations, although causes may be different, the system effects' manifestations may be similar.

Therefore, in this work, we will restrict our exploration of reliability, security, and derivation of quantifiable evaluation metrics to system state set analysis. It is fair to label this restriction as idealistic. Needless, we view this fencing of the problem a good starting point to explore these two concepts, reliability and security, from a mathematical analysis point of view.

Although, the formalism of finite state automata/machines under which computing systems are built is well understood, implementation considerations often betray this formalism. Attacks on these systems generally tend to exploit this gap between the formal specification and implementation. Practically, one cannot design an infinite state computing system. The state space is determined by the variables in the system which all have finite ranges, e.g., communication bus wires. Because of this physical implementation constraint, one can neglect to clearly separate these three aspects of the system: (1) formal specification—mathematical description of the system function, behavior, and properties; (2) functional specifications—capabilities and interfaces; and (3) functional implementation—hard or soft realization of the structure to carry out the function.

The organization for this work is organized as follows. Sections 2 and 3 provide the overview of computer system reliability and security, respectively. Section 4 summarizes the overlap of system security and reliability. In Section 5, we formalize the security and reliability in a unified generalized framework. Three case studies illustrate the proposed framework in Section 6. We conclude this work in Section 7.

2 Overview of Computer System Reliability

In this section, we explore the system reliability issues through the error models and solutions a reliable system usually is able to deal with. We will also study the limitation of those approaches. In this section, we refer the original source data and system states, as the information a system holds.

2.1 Brief Overview of Reliability

Due to the flimsy nature of the early computing systems, the concepts of the availability and reliability quickly became front and center design and specification issues. For example, the Harvard Mark I [1], built in 1944 in IBM Endicott laboratories, broke down once a week. The system weighed 5 tons and had 750,000 components. Consequently, the early definition of reliability in these systems took a very operational view. Reliability was defined as the conditional probability that the system would remain operational during the time interval $[0, t]$, given that it is functional at time $t = 0$. This definition has led to the widely used notion of *Mean Time to Failure (MTTF)* or *Between Failures (MTBF)*. The process that addresses system reliability is further broken down into few steps: (1) fault monitoring via detection; (2) fault mitigation through confinement, masking, and correction; and (3) fault recovery by system reconfiguration and state restoring.

2.2 Error Models for Reliability Problems

Since reliability cares mostly about a system being properly functional for a specific period of time, thus the non-invasive abnormalities, such as information leakage, will not be considered as a problem that concerns a reliable system.

Reliability issues are essentially loss of functionality problems. This usually appears as either the loss of data integrity or erroneous system state transitions. The two can be closely related to each other. For instance, a distortion of a variable value may lead to a wrong conditional jump, or inappropriate memory read may result in the change of data in memory cells.

2.2.1 Error Types

If we treat both the data and the state as the necessary information that a system holds in order to operate, then there are commonly two representative error types leading to the loss of functionality:

1. Missing of part of the information;
2. Distortion of the information.

Both can be spotted by error detection in a system. To restore the system functionality, the former requires information regeneration based on the remaining known

information, and the latter information correction, which is more challenging. Both can lead to either an error that can be handled by a reliable system, or a system failure.

2.2.2 Error Characteristics

Reliability does not address all the problems. For the errors that a reliable system can deal with, some of their common characteristics are:

1. Observable: the errors can be observed by comparing the distorted information with the original. Reliability rarely addresses information leakage problems.
2. Random: the errors should be caused by randomly generated faults due to the system's own instability (such as manufacturing flaws or aging), or external environment factors (such as unstable power supply, change of temperature, etc.).
3. Finite: the errors are usually bounded by limited magnitude and number. The reliability design of a system is usually based on the experience of the observed errors in the past, or the estimation of predictable instability.

In contrast, in a secure system, more problems should be taken into consideration: the errors can be carefully chosen and injected by the attackers, the attacks can be non-invasive such as side-channel, there may not be a limitation to the errors' frequency or multiplicity. The secure system designers have to go beyond the past experience and strive for the solutions for the more (or the most) grave situations a system can encounter.

2.3 Solutions for Reliability Problems

Altogether, the goals of a reliable system can be classified as (i) *error prevention*, (ii) *error monitoring*, and (iii) *error recovery* (cf., Sections 2.2.1 and 2.2.2). Generally, extra resources (storage, power, delay, hardware, etc.) can be added to the design to attain the abovementioned reliability features.

2.3.1 Error Prevention

One approach to achieve reliability is to prevent the system being harmed by its design flaws or instability before any fault could happen. For example, the hardened storage cells for memory [29] and hazard-free circuit for logic design can efficiently prevent some common faults or glitches from happening, such as single-event upset (SEU), single-event transients (SET), or other types of logical hazards.

2.3.2 Error monitoring

It is not uncommon for errors to happen in a system designed with error avoidance. Therefore, another important aspect of reliability design will be error detection. Usually this is attained by monitoring the system for the error types in Section 2.2.1 by both offline and online testings.

The offline testing uses a large set of random tests to apply to the system when it is not running any tasks (i.e., at idle state) and compares the results with the pre-stored reference results. Usually, exhaustive tests are hard to accomplish because of the limitations of cost and time. Therefore, the system designer and user should have an agreement on the trade-off between the desired test coverage and the cost.

The online testing runs the test during the system operation and generates real-time signatures to verify the correctness of the system's functionality. It requires additional hardware, software, or time redundancies in order to monitor the errors in operation.

Tests can be applied to different levels of a system, from as low as electronic components, circuits, and subsystem to system level. It helps a system discover its instability and builds confidence of its functionality.

One widely seen example of error detecting scheme is the duplication subsystem [25], where two systems with identical functionality will perform the same operations, and the results of which will be compared. An inequality indicates an error detected on either of the two systems.

2.3.3 Error Recovery

The greater need, other than detecting the errors, is to recover the functionality when errors truly happen to the system.

A self-checking or self-healing scheme needs to meet at least but not limited to these criteria:

1. It does not affect the functionality of the original system;
2. It generates real-time signatures corresponding to the system operations;
3. The signatures enable the system to detect and correct (or mask, mitigate) errors.

This scheme usually consists of a real-time signature generator and a verifier which takes both the original system's information and its corresponding signature for self-healing.

The triplication scheme is both an error detection and correction subsystem with fault tolerance. Instead of two

identically functional systems, it involves three in each computation task. One system's output can be viewed as the original information, the other two's the signature. A majority voting will be carried out among the three to tolerate the malfunction of a single system [23].

It is notable that in order to enhance the reliability of a system, except some approaches such as memory cell size increasing, most of the techniques hold the nature of linearity. This is because linear functions are easier to scale, compress, and reverse, making them befitting the demands of reliability-oriented designs in testing, error locating, information restoration, etc.

2.4 Limitations of Reliability-Oriented Designs

Although reliability-oriented designs can settle most random factor-caused problems, there are certain intrusions beyond its capability.

2.4.1 Excessive Random Errors

As stated previously, the design of a reliable system is usually based on the past experience of the most common observed errors. Therefore, when the random errors exceed the capability of the system, it will result in misdetection of errors or failure of system.

2.4.2 Injected Errors from Attackers

If a reliable system is designed with linear functions in order to restore its distorted integrity, then attackers can take advantage of it to inject invisible errors, which will never be spotted by the system. In addition, unlike security-oriented designs, the reliable designs usually have no secret (encryption key, digital signature, etc.) hidden from anyone, making it easier for the attackers to exploit the system.

The excessive random errors and injected errors from attackers usually can appear in a similar pattern: both could be invisible or uncorrectable to the reliable systems. How to draw a line between the two is worth discussion. One reasonable assumption one could make is that the excessive random errors should appear neither too frequently nor in a great density. A probability bound of the excessive random errors can be set such that once those errors are observed in a frequency or density above this bound, it can be categorized into attacks.

2.4.3 Non-Invasive Attacks

While a reliable system aims to maintain its functionality, it is not resistant to non-invasive attacks which do not affect the system operation, but only acquire system information stealthily. For example, eavesdropping, side-channel, or man-in-the-middle are all beyond what a reliable system can handle.

3 Overview of Computer System Security

3.1 Security Threats

The security of the system is mainly defined by confidentiality, non-repudiation, integrity, availability, and resilience to physical attacks. Confidentiality ensures that sensitive information is protected from unauthorized entities. Ensuring confidentiality is ensuring that those who are authorized to access information are able to do so and those who are not authorized are prevented from doing so. Non-repudiation is the assurance that an entity cannot deny its involvement in the action that it took part of. Integrity includes maintaining the consistency, accuracy, and trustworthiness of data or resources. To maintain the integrity, data and resources cannot be modified by the adversary or third party without authorization of a legitimate user. Availability refers as access to the information and other important resources without undue delay and on demand by an authorized entity. To maintain the availability, the system should perform the rigorous maintenance of the hardware and system resources to ensure the smooth operations. Physical attacks like bus probing, timing analysis, fault induction, power analysis, and electromagnetic analysis have been demonstrated to be powerful in easily breaking the security.

In this work, we will focus on integrity and availability attacks as these attacks share common ground for the reliability and security.

3.1.1 Integrity and availability attacks

The attacker's objective can range from stealing private information (confidentiality/privacy), destroying the system (availability), or tampering the system states or data (integrity) for other than its intended purpose.

Integrity is compromised when the intruder modifies or deletes the important data or tampers the resources to fulfill his/her malicious intentions. Availability attacks disrupt the normal functioning of the system by misusing system resources so that they are not accessible for normal operation. Integrity and availability attacks require interfering with the system in some manner; hence, they need active participation of the attacker. We listed the following attacks which harm the integrity and availability of the system.

Integrity and availability objectives can be thought of as common grounds for security and reliability of the system. These objectives essentially spoke for the security aspect of the systems.

1. **Man in the middle (MITM) attack:** MITM attack is one of the most common type of security attacks [11] which threatens integrity and availability of the system. MITM attack risks the integrity by intercepting the communication between two end points and modifying the data. It also threatens the availability of communication by intercepting and destroying data or modifying data to seize the communication.
2. **Denial of service (DoS) attack:** The goal of the DoS attacks limit the availability of the system.
3. **Attacks due to Virus/Trojan horse/Worms:** A malicious software may carry virus, Trojan horse, or worms. A common feature of these malware is that they all have undesirable, potentially harmful functionality which will infect the system in a destructive way. These attacks can manipulate the sensitive data or processes harming the integrity as well as denying access to system resources threatening the availability.
4. **Fault injection attack:** Fault injection attacks rely on changing the external parameters and environmental conditions of a system and can be performed by deliberate injection in system's component with the help of white light, laser beam, voltage/clock glitch, and temperature control, radiation, etc. [3]. Fault injection attacks can obstruct availability of the system. Faults can be inserted in the system to disturb the normal function, e.g., set the bus lines to constant value [33]. This attack can cause integrity concerns where adversary can inject the faults to corrupt the data or code stored in the system components like memories.
5. **Hardware Trojans:** Hardware integrity can be challenged through malicious additions, i.e., hardware Trojans (HTs) in Network on Chip (NoC) [7]. The inserted hardware Trojans can pose Denial-of-Service (DoS) threats which will cause network degradation and eventually availability issues in the systems. Hardware Trojans can be used to inject faults [44] to further reduce the integrity of the device.

Figure 1 shows an illustration of these attack domains and their grouping.

3.1.2 Confidentiality Attacks

Although there do exist invasive attacks to break a system's confidentiality, such as injecting errors to AES stages in order to steal the secret key, confidentiality attacks are usually non-invasive. Unlike integrity and availability, confidentiality is a problem that usually does not concern reliability-oriented systems since it does not affect the functionality of a system. A system can be reliable in carrying out its operations, but leaking important information to the attackers at the same time. Some of the ways to break a system's confidentiality are:

1. **Eavesdropping:** for example, MITM can intercept the information sent between two terminals;
2. **Side-channel:** for example, power analysis;
3. **Unauthorized readout:** for example, reading out the unprotected content in memory cells.

3.2 Existing Hardware Infrastructures for Security

The existing system security features rely on protecting the system components by modifying the hardware infrastructure. In computing devices, traditional operating systems and the services are becoming so large and complex that the task of securing them is increasingly harder. To fill this gap, the hardware-based trusted execution environments (TEEs) were developed. TEE on a device is isolated from its main operating environment by using hardware security features. The trusted execution environment can be realized in different ways, but the overall concept stays the same. Hardware-based trusted execution environments offer isolated execution, secure storage, remote attestation, secure provisioning, and trusted path. However, none of the TEE considers the reliability concerns at the system level. With-

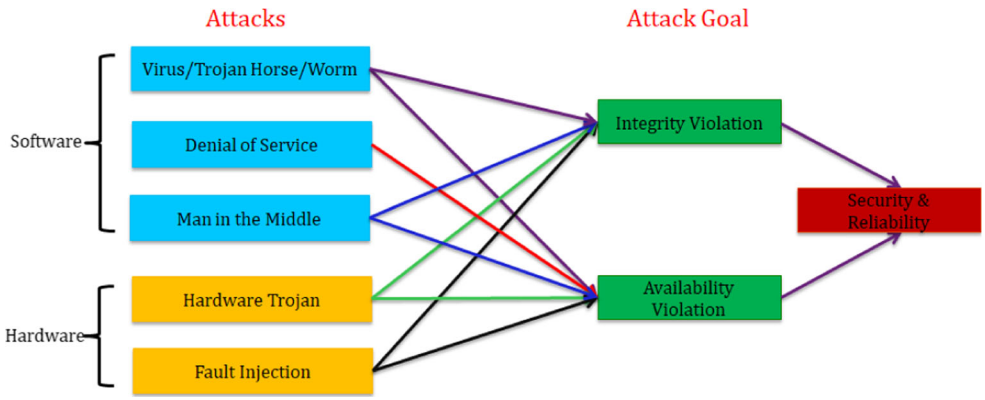


Fig. 1 Integrity and availability attacks

out the awareness of the impact from unreliability, the system security will be undermined. Some of the TEEs are summarized below:

The ARM TrustZone [2] for enable trusted computing is based on a trusted platform, wherein a hardware architecture supports and implements a security infrastructure throughout the system, rather protecting dedicated hardware block. The TrustZone methodology is the concept of secure and non-secure worlds. These worlds are separated by the hardware, wherein non-secure software do not have direct access to secure resources.

Intel's Software Guard Extensions (SGX) [12] protect selected code and data from modification or revelation. The trusted hardware establishes enclave (secure container) and the remote computation service. The user uploads the desired computation and data into the enclave. Enclaves are protected areas of execution. The trusted hardware protects the confidentiality and integrity of data under process.

The Capability Hardware Enhanced RISC Instructions (CHERI) architecture [42] extends the commodity 64-bit MIPS Instruction-Set Architecture with new security primitives. These primitives help the software to efficiently implement fine-grained memory protection and an object capability security model. CHERI adds a new capability coprocessor that supports granular memory protection within address spaces in existing RISC CPU.

LowRISC is a fully open-source SoC based on the 64-bit RISC-V instruction set architecture. In the lowRISC- tagged memory implementation, every memory address is appended with tag bits. In this way, the lowRISC implementation supports basic process isolation. To protect against memory corruption of the return address, table pointers, and code pointers on the stack/heap, the compiler is modified to generate instructions to mark vulnerable locations.

Reliability and security have emerged as two distinct directions for many years. However, reliability and security can be strongly interdependent: one can be a condition to the other and they share many commonalities. Even though the abovementioned TEE can stop some of the security threats at the system level, they do not address the underlined issues related to reliability. If the reliability concerns can not be considered while designing the security objective, the security can be compromised.

4 System Reliability and Security

4.1 Interleaving of Reliability and Security Issues

The positive and negative impacts of reliability issues on the circuit/system security are coexisting. Process/voltage/temperature variation phenomena and the causes of unreliability have been widely leveraged to design security primitives. Thus, we do not necessarily require managing the reliability challenges at the physical level, as those unreliable features may be exploited to strengthen the resistance against security attacks. Measurements from device aging tests provide a good clue to identify counterfeit chips. This is another benefit we can obtain by leveraging reliability study. Hardware Trojan horse is one of the negative impacts from the reliability domain.

4.1.1 Leveraging the Sources of Unreliability to Design Security Primitives

The design of well-known physical unclonable functions (PUFs) [16, 17, 21, 35] relies on the unpredictable physical characteristics that are unique to each fabrication process. PUFs receive increasing attentions due to its theoretical unpredictability and low cost for the purpose of hardware metering. In contrast, the presence of process variation typically requires chip designers to increase the design margin so that the potential process variation-induced reliability issues can be tolerated.

Noises originating from transistor thermal activities, power supply, clock jitter, and the discrete nature of electric charge (i.e., shot noise, flicker noise) are exploited as entropy sources to generate random bits through random number generators (TRNGs) [10, 24, 28]. If noise mitigation methods are applied to the system, the entropy that is beneficial to TRNGs will be eliminated as well. A system designer should carry the awareness of the coexisting measures for reliability and security.

4.1.2 Leveraging the Sources of Unreliability to Detect Counterfeit Chips

Bias temperature instability (BTI) [18], electromigration (EM) [39], hot carrier injection (HCI) [37], and time-dependent dielectric breakdown (TDDB) [8] are the common reasons to cause device aging or to wear out. The BTI, EM, HCI, and TDDB mechanisms used to be the interest for the reliability community that are now helping security community detect the counterfeit chips. The counterfeit chips may be induced from the one who recycles the worn-out devices back to the chip market. The testing methods aiming for BTI, EM, HCI, and TDDB detection have been utilized to detect counterfeit chips. However, the methods that can extend device lifetime may not be ideal to prevent the counterfeit devices from returning back to the IC market.

4.1.3 Leveraging the Sources of Unreliability to Hide Hardware Trojan Horses

Hardware Trojan horses refer to malicious modification to the original hardware design, from netlist to layout

level. To evade testing-based screening, a large number of hardware Trojans activate the Trojan trigger by accelerating the speed of hardware wearing out, rather than waiting for the arrival of a specific logic input combination. Another type of Trojans, i.e., parametric Trojans, is created during the fabrication stage, where the physical geometry of a transistor or a metal wire is altered. The changes on transistor width, dopant concentration, and interconnect pitch will lead to some phenomena similar to device defects. Trojan horses based on unreliable causes will confuse the reliability measures. Typically, methods for reliability provide the optimal protection regarding random defects, but those methods may not be sufficient to combat the "defects" induced by security attacks, which may be in burst and consistent. If the methods for reliability are not designed with special caution for upper limits, the system may be interrupted too often by security attacks, thus seriously affecting the system performance and wasting resources.

4.1.4 Trade Off or Collaborate for Reliability and Security Need?

When we treat unreliability causes, we may take advantage of those phenomena for security purpose. So, it is a trade-off. A system designer should be aware of coexisting measures for reliability and security (Fig. 2). Enhancing the reliability may lose the system's attack resistance. Methods that are *externally* applied to the device/circuit to assure reliability can be re-used for some security purposes, for instance, counterfeit chip detection. However, the mechanism *built into* the device/circuit (at least low physical level) will expand the live space for counterfeit chips.

4.2 Illustrative Case: Rowhammer Attack

High-density DRAM is more likely to suffer from disturbance in which different cells interfere with each other. Due to the closeness of cells, the voltage fluctuations while activating a row are more likely to affect adjacent rows and finally result in bit flips. This phenomenon of causing the disturbance is called Rowhammer attack [15, 27]. Several studies [15, 19, 32, 36] have observed that Rowhammer can be exploited to mount an attack and bypass most of the established software security and trust. This attack has been identified as a *security* and *reliability* threat [15, 32]. By repeatedly accessing the same memory row (aggressor row), an attacker can generate enough disturbance in neighboring rows (victim rows) to cause a bit flip.

The Rowhammer DRAM bit-flipping attack [5, 19, 34, 36] is an example of a different class of software attacks that exploit design defects in the computer's hardware. In [27], authors demonstrated that malicious software can take advantage of the CLFLUSH instruction, which flushes the cache line that contains a given DRAM address. CLFLUSH is envisioned to extract better performance out of the cache hierarchy and is available to software running at all privilege levels. Seaborn et al. implemented Rowhammer exploits [36] in native code with the CLFLUSH instruction: a privilege escalation on a Linux system caused by a bit flip in a page table and an escape from the Google Native Client sandbox caused by a bit flip in indirect jumps. The method in [5] combines knowledge of reverse engineering of last level cache (LLC) slice and DRAM addressing with timing side-channel to determine the bank in which the secret key resides. In this approach, timing-based technique is used to learn where the secret keys are stored in the bank in order to flip their bits and then activate the bug on the specific bank and to produce a flip bit of the row's secret data. The Rowhammer attacks target special memory management features deduplication (an OS feature widely deployed in production) and virtualization [34, 43]. Memory deduplication allows an attacker to reverse-map any physical page into a virtual page if he/she knows the page's contents. Hence, the OS can be tricked to map a victim owner memory page on the attacker-controlled physical memory pages. In [43], authors exploited cross-VM settings where bit flips are induced by Rowhammer attacks to crack memory isolation enforced by virtualization. This type of exploit bypasses password authentication on an OpenSSH (SSH: Secure Shell) server.

5 Security and Reliability Formulations

In this section, we introduce the SRASA (*security and reliability aware state automaton*) model of computation, which is an abstraction of the system transitions among the unprotected state, reliable state, secure state, vulnerable

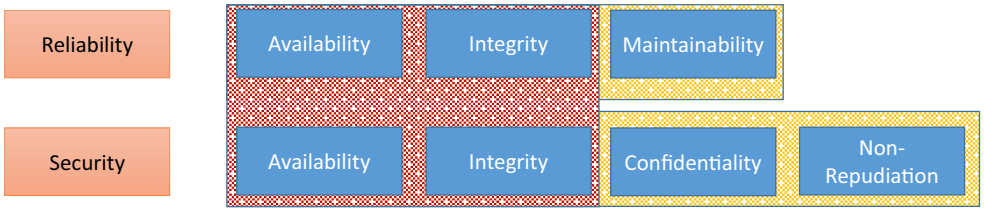


Fig. 2 Reliability and security objective overlap

state, restored state, etc. This abstraction helps the system designers achieve a trustworthy design, or the testing team to evaluate a given design or implementation under desired reliability or security (or both) criteria.

SRASA treats any system as a state machine. The state machine consists of trusted states, unknown states, vulnerable states, etc. SRASA defines multiple functions, which transit the state machine from one type of state to another. These functions are equivalent to the approaches preventing or restoring a system from random errors or malicious attacks, or instability and vulnerability caused by hostile environments. The specified functionality and data processing behaviors of the system are the alphabets and operations on the states.

With this mathematical abstraction, SRASA provides a systematic approach for reasoning about the trustworthiness of the system under design. The capability to maintain a system in the reliable or secure states, or restore a system from the vulnerable states, can be evaluated by both system designers or testers, thus making improvement to the system.

Figure 3 shows how SRASA packages and unpacks a system figuratively.

However, it is worth noting that the user of the system ultimately only cares about its functionality, not the details in achieving reliability and security. Thus, the reliability- or security-oriented designs should not impact the system's original functionality.

In the following subsections, we formally introduce our proposed security and reliability aware state automaton.

5.1 Preliminary Definitions

One of the general assumptions often made is the state space of the three aspects of the system perfectly fits. Although the functional implementation defines this finite set, both formal specification and functional specifications could define infinite sets—this is the main principle around which strong encryption is built. In fact, the overlay of these state space creates three distinctive sets of states that we denote in the work as X_r, X_s, and X_v.

X_s is a countable set of states that implements the system's functional specifications. X_r is a countable set of states in the system that falls outside the functional specification, but is covered by the formal specification and therefore tested for reliability issues. X_v is a countable set of vulnerable states in the system that are implementation by-products; they could even fall outside both formal and functional specifications. X_v includes the vulnerable states that a system can be driven into due to reliability issues. X_v is the easiest set of states for an attacker to push the system into to bypass design and development analyses.

In terms of cardinality of the state space, we have the following:

$$n(X_s) \leq n(X_r) \leq n(X_v) \tag{1}$$

Where $n(A)$ denotes the cardinality of the countable set A.

As shown in Fig. 4, X_r is a set of states outside of the X_s set and X_v is outside both X_s and X_r. A state classified under X_s is simply deemed secure as per the functional specification. A user may specify an unsecured/vulnerable or unreliable state to minimize system complexity, cost, or deployment constraints. Not explicitly highlighted on the figure for simplicity is the set of states at the intersection of X_s and X_r. We denote these states X_d as the set of highly dependable states (both reliable and secure).

The notion of dependable set of states (X_d) focalizes the desired properties of next-generation computer systems, where reliability and security features are seamlessly integrated. An illustration of this notion is the case where a customer orders an antique vase online. The vase is wrapped with bubble wraps to protect it from damage, attached with a certificate of authenticity to prove it is

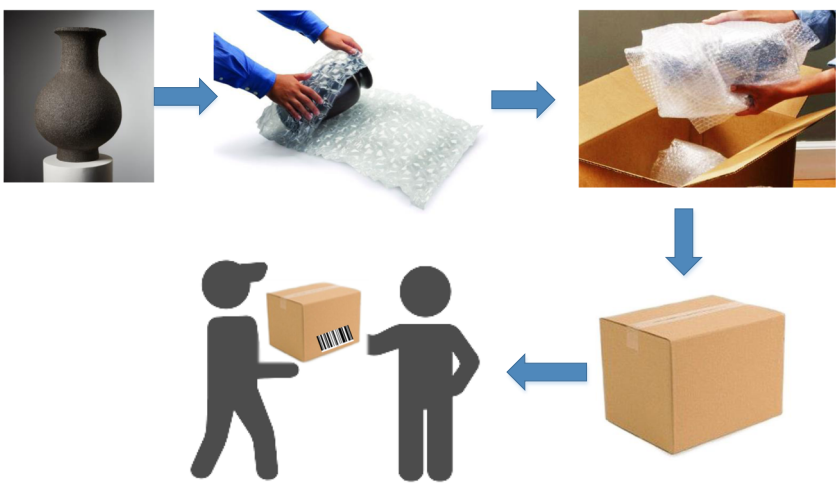


Fig. 3 A figurative illustration of SRASA: A trustworthy purchase (design) of a vase (a system) is firstly protected by bubble wraps against possible damage (reliability against random errors). Then, it is put into a box to preserve privacy (security against non-invasive attacks). There may be additional shipping services to ensure authenticity and non-repudiation, i.e., at the receiver (user) end, the customer checks the sending certificate (security against tampering or spoofing) to verify its authenticity and signs it to acknowledge the delivery

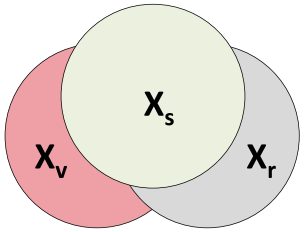


Fig. 4 Functional set of states X_s, deemed reliable set of states outside functional states X_r, and vulnerable states that fall outside functional and reliable states X_v

not a counterfeit, and packed with a non-transparent box for privacy. This parcel covers both reliability (danger of damage) and security (counterfeit, loss of confidentiality) issues. However, the customer only cares that the genuine antique vase is delivered intact and safe, rather than how the store has packed it. Similarly, a computer system user should mostly care about the functionality Σ_l of the system. On the other hand, in addition to the functional specification of system, designers should consider its secure and reliable properties.

One key deduction of the theoretically infinite state behavior of computing systems is (1) states could be countable infinite—i.e., we know at every point/state how (rules under which) the system transitions, and yet we do not know all the states; and (2) guardian or guard state is the state up to which system behavior/state space (states and state transitions) is defined, fully explored, and tested. Between the initial and guard states, the system properties are fully defined: (a) determinism, (b) closure/boundedness, (c) convergence, (d) observability, and (e) coverability. Another direct implication of this concept is the initialization of state of the system.

5.2 Security and Reliability Aware State Automaton

Formally, SRASA model of computation is a 22-tuple state machine $(\Sigma_l, \Sigma_{ia}, \Sigma_{ip}, X_s, X_r, X'_s, X'_r, X_v, X_d, f_s, f_r, f_v, f_{sr}, f_{rs}, f_{rv}, f_{vr}, f_{vs}, f_{sv}, x_0, F_s, F_r, Fv)$ where Σ_l is a countable set of legal alphabet symbols—from a system point of view, an alphabet symbol is an input datum or data or action like "power on," "start," and "stop";

Σ_{ia} is a countable set of illegal active alphabet symbols—an active alphabet symbol is an input that manipulates either data or state processes in the system;

Σ_{ip} is a countable set of illegal passive alphabet symbols—a passive alphabet symbol is one that reads either main output data of system, i.e., through direct system I/O or secondary outputs like "heat dissipation";

X_s is a countable set of states that implements the system's functional specifications—X'_s is the mirror set of X_s;

X_r is a countable set of states in the system due to implementation glut with no functional use, but tested for reliability issues—X'_r is the mirror set of X_r;

X_v is a countable set of vulnerable states in the system—these are states where an attacker will try to push the system into;

X_d is the dependable set of states and represents the intersection of X_s and X_r sets;

f_s is the state transition function for secure system execution: $f_s : X_s \times \Sigma_l \rightarrow X_s$;

f_r is the state transition function for reliable system execution: $f_r : X_r \times \Sigma_l \rightarrow X_r$;

f_v is the state transition function for unsecure system execution: $f_v : X_v \times (\Sigma_{ia}|\Sigma_{ip}) \rightarrow X_v$;

f_{sr} is the state transition function that moves the system execution from a secure state to a dependable state: $f_{sr} : X_s \times \Sigma_l \rightarrow X_d$;

f_{rs} is the state transition function that moves the system execution from a reliable state to a secure and reliable state: $f_{rs} : X_r \times \Sigma_l \rightarrow X_s$—a more conservative function will be $f_{rs} : X_r \times \Sigma_l \rightarrow X'_s$;

f_{rv} is the state transition function that moves the system execution from a reliable state to a vulnerable state: $f_{rv} : X_r \times (\Sigma_{ia}|\Sigma_{ip}) \rightarrow X_v$;

f_{vr} is the state transition function that moves the system execution from a vulnerable state to a reliable state: $f_{vr} : X_v \times (\Sigma_{ia}|\Sigma_{ip}|\Sigma_l) \rightarrow X'_r$;

f_{sv} is the state transition function that moves the system execution from a secure and reliable state to a vulnerable state: $f_{sv} : X_s \times (\Sigma_{ia}|\Sigma_{ip}) \rightarrow X_v$;

f_{vs} is the state transition function that moves the system execution from a vulnerable state to a a secure and reliable state: $f_{vs} : X_v \times (\Sigma_{ia}|\Sigma_{ip}|\Sigma_l) \rightarrow X'_s$;

This formulation encapsulates the notation that a reliability bug or an attack introduces no new state to

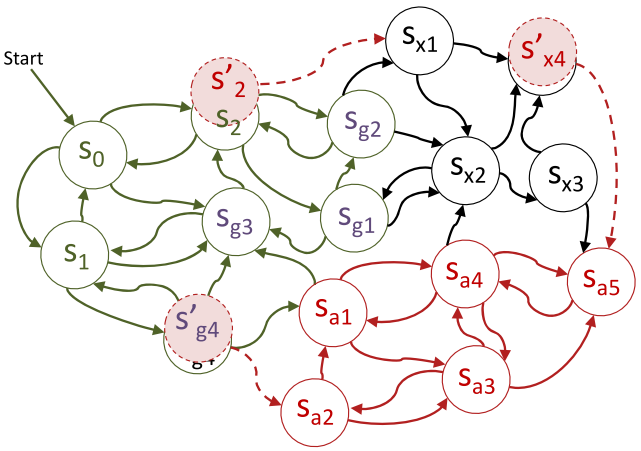


Fig. 5 Security and reliability aware state automaton (SRASA) overview

the computing system. An attack or reliability bug simply transitions the system from a well-specified state to a less known or desirable state (Figs. 5 and 6).

5.3 Positive and Negative Functions

We categorize the functions which implement the reliability and security as positive functions. These functions are usually added by the system designer at the design stage, or carried out by the system at its operation.

1. During system design or operation: f_r, f_s;
2. When faults, errors, or attacks occur: f_{vr}, f_{vs};
3. To enhance the system with addition feature: f_{rs}, f_{sr}.

It should be emphasized that there is an important functional distinction between f_{rs} and f_{sr} state transitions, although they both may lead to a dependable state. These state transitions are as follows:

$$f_{rs} = f_r(f_s);$$
$$f_{sr} = f_s(f_r).$$

Although f_{sr} and f_{rs} both provide both security and reliability to the system, their order of applying reliability and security functions to the system is different. Consequently, the security or reliability guarantees of the system may be vastly different. For example, to render a piece of data resilient to both eavesdropping and random errors, the correct order is to first encrypt it (e.g., using AES), then encode it with some ECC technique. If the piece of data is encoded before encryption, then there is no protection against random errors, since altering a single bit of the cipher text may change all the bits in the decrypted plain text, making it uncorrectable by the ECC technique applied.

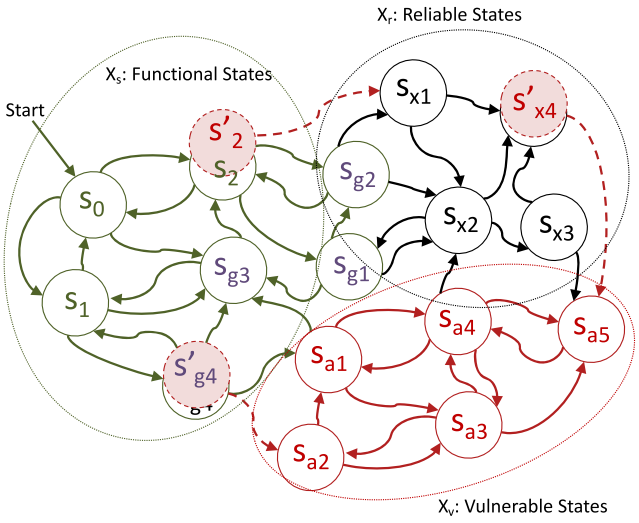


Fig. 6 SRASA state partitioning

We categorize the functions which attempt to harm a system's functionality as negative functions. These functions are applied by either environmental or artificial abnormality.

1. To attempt to harm an unprotected system: f_v;
2. To attempt to harm a protected system: f_{rv}, f_{sv}.

5.3.1 Relationship Between These Functions

If f_{vs}, f_{vr} are to restore the system to its normal functionality from an erroneous or insecure state, then we may have the following relationships:

$$f_{vs} = f_s^{-1}, \tag{2}$$
$$f_{vr} = f_r^{-1}. \tag{3}$$

The $^{-1}$ superscript does not stand for a strict mathematical reverse. It means functionally to trace back, restore, or verify the right-hand side functions.

For instance, f_{vr} can be the decoding function for random error tolerance in a system protected by an encoding function f_r at its operation. And, f_{vs} can be the authentication function to detect any tampering of a system's data securely hashed by f_s.

5.4 Illustration of the Functions in Phases

In this section, we will show how the functions affect a system in different operation phases. All these functions are state-transitioning functions, with the help of alphabets, either legal ones or illegal ones. If the positive functions can achieve their mission to stabilize and secure the system against negative functions, then we should be able to have the following relationships between them:

- **Phase 1 - Preparation:** Moving the system to a reliable or secure state by positive functions. It needs to be done before the system is exposed to errors or attacks.

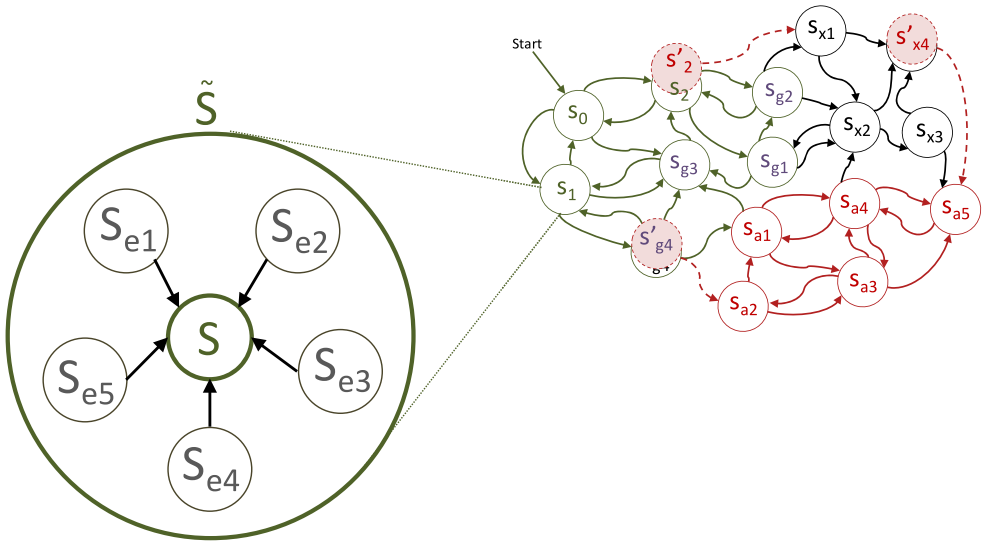
$$f_r(\Sigma_l) = X_r; \tag{4}$$
$$f_s(\Sigma_l) = X_s. \tag{5}$$

Examples encoding, signing, or encrypting piece of information before transmission, duplicating, or triplicating a system before running a task, etc.

Besides, f_r and f_s can also be treated as the protection added in the design stage, that the designer of a system decides to integrate certain redundant submodules in order to empower the system with reliable and secure features.

- **Phase 2 - Intrusion:** Moving the system to a vulnerable state by negative functions through random errors (Σ_{ia})

Fig. 7 SRASA reliability view



or attacks ($\Sigma_{ia}|\Sigma_{ip}$). It happens during the system operation.

$$f_{rv}(X_r) = X_v; \tag{6}$$
$$f_{sv}(X_s) = X_v. \tag{7}$$

Examples: random errors, aging, injected errors, eavesdropping, man-in-the-middle, DOS attack, etc. These faults, errors, or attacks will tempt to harm the functionality of a system, in an either invasive or non-invasive manner. They can be handled by the system's reliability or security submodules prepared in Phase 1, or they can lead to a system failure.

- **Phase 3.a - Restoration:** Moving the system back to a reliable or secure state by positive functions. It happens after the system is distorted, either actively or passively.

$$f_{vr}(X_v) = X_r \text{ or } X'_r; \tag{8}$$
$$f_{vs}(X_v) = X_s \text{ or } X'_s. \tag{9}$$

Examples: error correcting or authenticating a distorted message, or checking the timestamp of an operation against replay attacks.

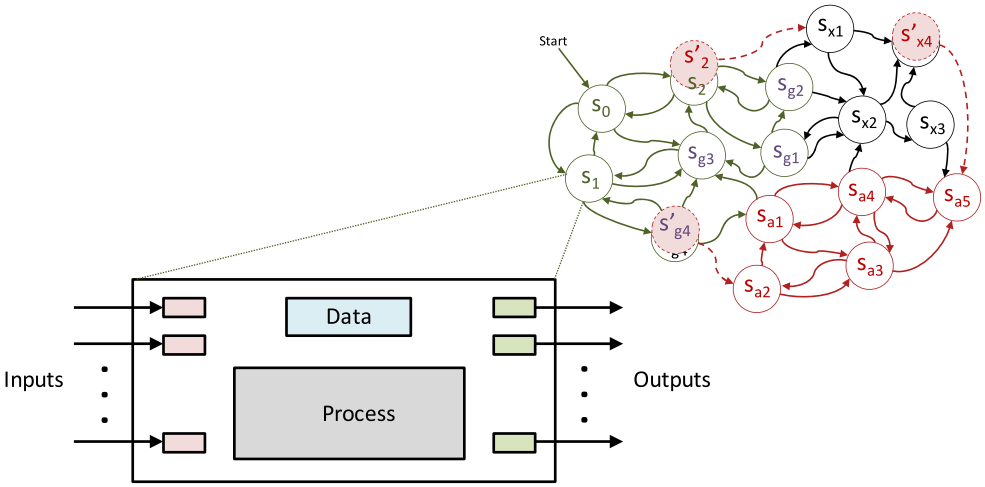
- **Phase 3.b - System Failure:** System unable to be moved back to a reliable or secure state by positive functions.

$$f_{vr}(X_v) = X_v; \tag{10}$$
$$f_{vs}(X_v) = X_v. \tag{11}$$

Examples: errors exceed the system error tolerance capability, a successful collision attack to authentication, a 0-day exploit on the vulnerabilities unknown to the designer, etc. For these types of errors or attacks, the system either cannot detect handle it, leaving the system always stuck at the X_v state.

Now, it is worth to fully describe what is a mirror state. In a mirror state, consisting of internal state data and processes, that looks identical to the state, both the contexts of execution have changed. Either the system has

Fig. 8 SRASA system view



transitioned back into the state after visiting a vulnerable or weak reliability state or it has sustained a *non state modifying* attack, i.e., a passive attack, e.g., a side-channel attack. In the X_s, we have the subset X_g. This is the set of guard states.

SRASA provides formalism for key established security features. For example, the transition from a weak reliability or vulnerable state to a mirror state in the X'_s provides *non-repudiation* guarantees. Similarly, the mirror states themselves give the formulation integrity checking support.

Figure 7 shows the reliability view of the SRASA formulation. An attempt to classify or count all the state will not be practical, even for the set of states deemed secure and reliable. Therefore, the reliability view is to anchor key states and use conventional reliability techniques (e.g., error detection and correction) to coalesce states born out of small and normal system variations into these anchor states. In essence, this is one of the main functions of reliability support in a computing system. Figure 8 depicts the concrete system view of the states.

Although our definitions of system, data, and process are simple, they are complete in description a computer system of any scale and complexity and its operation. For example, the process definition is compositional as shown in Fig. 9. It is both clocked synchronous (e.g., *Processes A' and A"*) and asynchronous (e.g., *Process A"'*). It supports both parallelism (e.g., *Processes A' and A"*) and sequence compositions (e.g., *Processes A' and A"'*). The feedback loop operation is illustrated in *Process A"*.

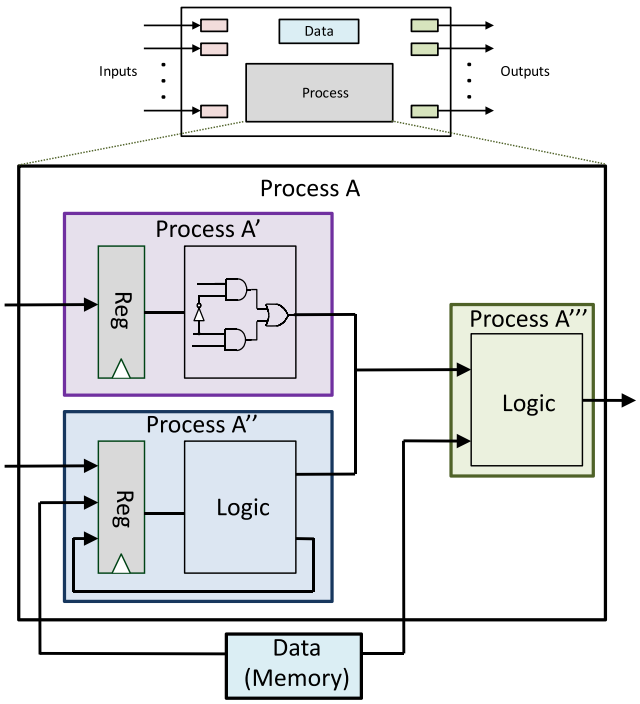


Fig. 9 SRASA system process view

5.5 Applying SRASA to the System Design and Evaluation Processes

A general yet complete methodology that will encompass all aspects of system design, from the functional level specification to the gate level design at any system granularity, may not be feasible or it may be beyond the scope of a single work. Therefore, we aim in this work to introduce a generalized framework to specify and reason about both reliability and security in the system design process. SRASA is a mathematical abstraction of a system's transition among different states (unprotected, vulnerable, restored, etc.). It is general enough to be adopted or customized for different specific and concrete design instances.

5.5.1 SRASA for System Design and Testing

For SRASA's application to specific system designs, different flows or procedures can be espoused. One such design flow is presented in [45]. In the research, the authors proposed a four-step design flow based on the SRASA concepts. Firstly, the designers can choose a reliable

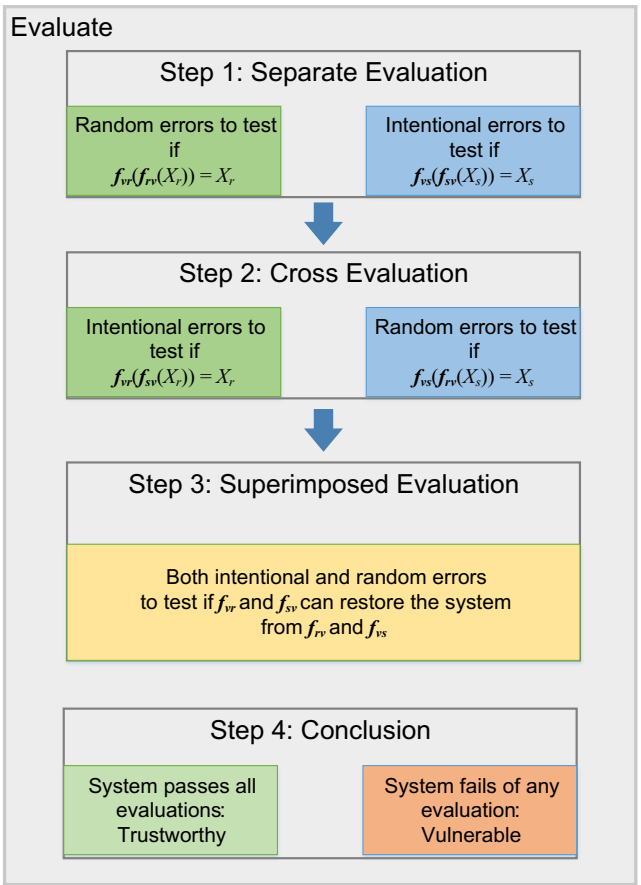


Fig. 10 The test procedure verifies if a given system is trustworthy on both reliable and secure issues

function f_r and secure function f_s based on the desired system trustworthiness. Then, pitfalls of f_r under security vulnerabilities and side effects of f_s to create reliability issues should be studied. Based on the step above, the designers can determine if f_r and f_s have overlaps or need a trade-off. In the end, various decisions can be made such as merging two methods, adjusting two methods, or no joint effort needed.

Another example of SRASA-based procedure to test and evaluate a given system is shown in Fig. 10. In this procedure, firstly, the reliability and security features are tested separately with known faults. Then, a cross evaluation is carried out to examine if the intentional attacks will go beyond the system's error tolerance capability, or if random errors will jeopardize its security capability. This step is necessary because there have been many proposals to inject "invisible" errors to bypass a system's reliability module, and approaches to acquire important information by side-channel attack from a system when random errors affect the outputs. Then, both random and intentional faults should be applied to the system to test its capability of restoration under grave situations.

5.5.2 Scalability and Complexity of the SRASA Formulation

The SRASA framework supports different abstraction views of the system. To curtail (1) state explosion or (2) countably infinite state set problems, one can fuse certain states together as a way to reduce the number of states to explore or to focus on specific sets of states deemed important or sensitive. The state-based approach of the SRASA formulation also lends itself to provable and systematic state transformations and reductions. For example, one can use the concept of *non-distinguishable states* to combine states that cannot be distinguished from one another for any input from the alphabet. These transformations have been shown to (1) maintain functional and correctness correspondence [6, 22] and (2) be performed in $O(knlogn)$ time complexity using $O(\frac{n}{logn})$ processors [38, 40]. The guard states S_g are also interface states and can be used fusion boundaries. Figure 11 illustrates such state fusion operations.

The following section provides practical and detailed examples on how SRASA can be used to guide the design and testing of a system.

6 Illustrative Design Cases of the Proposed Framework

In this section, we provide three case studies to illustrate the practical meanings for the states formulated in Section 5. Three case studies have different focuses: Case study 1: how the linearity of a reliability-oriented design can be leveraged by attackers; Case study 2: new system states reflected on hardware cost and power consumption; Case study 3: passive attack as an external force to trigger the system state transition, respectively.

6.1 Case Study of Error Detection Schemes for the S-Box of AES

Firstly, we present an example to use SRASA on testing and comparing two different designs to detect errors in the outputs of the S-BOX in AES. We show how the first design's linearity can be easily leveraged by an adversary [31], which is not the case in the second design.

Table 1 lists the physical meaning of each variable to formulate the problem.

For convenience, some terms are defined as follows:

- $a_{i,j}$: the element located at the i^{th} row and j^{th} column of v;
- $b_{i,j}$: the element located at the i^{th} row and j^{th} column of u;
- b: the number of bits in a byte;
- $GF()$: the Galois finite field;
- $\otimes$: the finite field multiplication;

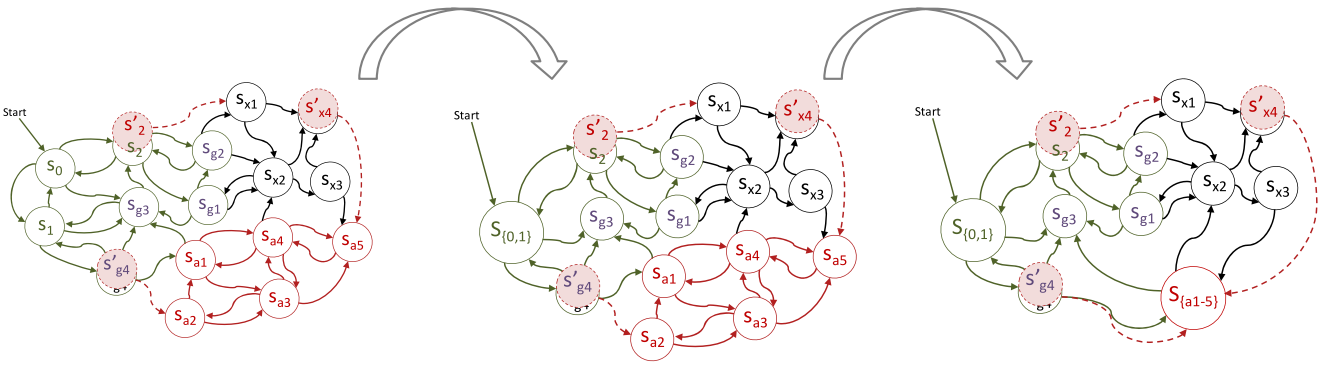


Fig. 11 SRASA state representation fusion operations. The automaton representation goes to S_0, S_1 states depending on the system input symbol, but they can be fused to an equivalent single state, giving the automaton a smaller set of states

Table 1 Terminology meanings in this S-Box case study

Symbol	Physical meaning in this study
$\sum l$	Inputs to the S-Box
$\sum ia$	Additive faults or injected errors to the S-Box
X_s	S-Box properly functional under secure protections
X_r	S-Box properly functional under reliable protections
X_v	Errors propagate to the outputs without being detected

- $\oplus$: the finite field addition;
- f_{SubBytes}: the SubBytes function;
- e: the additive error injected by attackers;
- $\sim$: the distortion symbol, e.g., $\tilde{b}_{i,j} = b_{i,j} \oplus e_{b_{i,j}}$.

6.1.1 The S-Box (SubBytes) Stage in AES

The SubBytes is the only stage which provides non-linearity to AES. It is usually referred to as the S-box. The S-box works over the Galois finite field of $GF(2^8)$ on each byte of the state. It firstly applies a multiplicative inverse to an element of v over the finite field, and then an affine transformation. Although the S-box is usually implemented by a 16×16 lookup table, it can be abstracted to the mathematical function below:

$$b_{ij} = f_{\text{SubBytes}}(a_{i,j}) = M_{inv} \otimes a_{i,j} \oplus M_{aff}, \tag{12}$$

where matrices M_{inv} and M_{aff} perform an affine transformation to $a_{i,j}$ over $GF(2^8)$.

6.1.2 To Protect S-Box with Reliable or Secure Functions

There have been many research efforts which have proposed protection schemes for the S-Box (and other stages of AES) using the self-checking checkers (SCC) based on error control codes (ECC). The common ground of these techniques is to add a parallel module named the "Predictor" to the original function module, which generates the corresponding check bits at the same time when an AES stage generates its output using f_{SubBytes}. Then, both the output of this stage and its check bits are verified by a "Decoder" for error detection, or correction. Together, the Predictor and Decoder form an SCC system.

Figure 12 illustrates the functional blocks of the SCC system. f denotes the S-Box function (12), the predictor function is represented by $P()$, the predictor's output is R_b, and the decoding function is $H()$.

Here, two designs based on Fig. 12 are evaluated. One using Hamming codes to build $P()$ and its corresponding $H()$ for the SCC, and another using a security-oriented Robust codes. The Hamming codes require at least 4 bits of redundancy produced by R_b, and Robust codes 8 bits. Therefore, there can be 2^{12} possible errors for the Hamming codes-based design, and 2^{16} errors for the Robust codes version.

Since Hamming codes are mostly used to address reliability issues, we denote such protection over S-Box as using a reliable function f_{vr}, and so Robust code using f_{vs}. If they are able to function properly under $\sum ia$, then we have the following equations:

$$f_{vr}(\sum l + \sum ia) = X_r \tag{13}$$

$$f_{vs}(\sum l + \sum ia) = X_s \tag{14}$$

Based on the suggested evaluation flow in Fig. 10, both designs will be examined by random errors, intentional errors, and both. Here, we have found that although the predictor based on Hamming code has the capability of detecting all single and double bit errors, while correcting all single bit errors (SEC-DED), there is a fatal vulnerability to it: invisible errors.

Due to the linearity of the Hamming codes based SCC, $P(f())$ and $H()$ are all linear functions. Particularly, they are matrix multiplications. Thus, we have:

$$H(\tilde{R_b}, \tilde{b}) = H(R_b \oplus e_{R_b}, b \oplus e_b) = H(R_b, b) \oplus H(e_{R_b}, e_b). \tag{15}$$

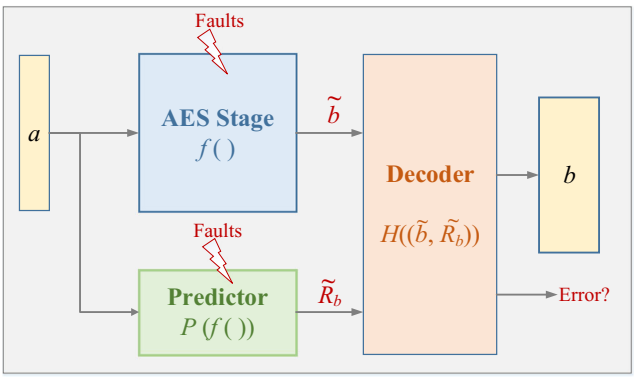


Fig. 12 The predictor applies a combined function of $P(f())$ over the state element a, which will result in $f(a) = b$'s check bits. Both b and R_b could be erroneous and will be verified by the decoding function $H()$. If $H()$ is evaluated as 0 by vector $(\tilde{b}, \tilde{R_b})$, then it is considered as error free, otherwise an error detected

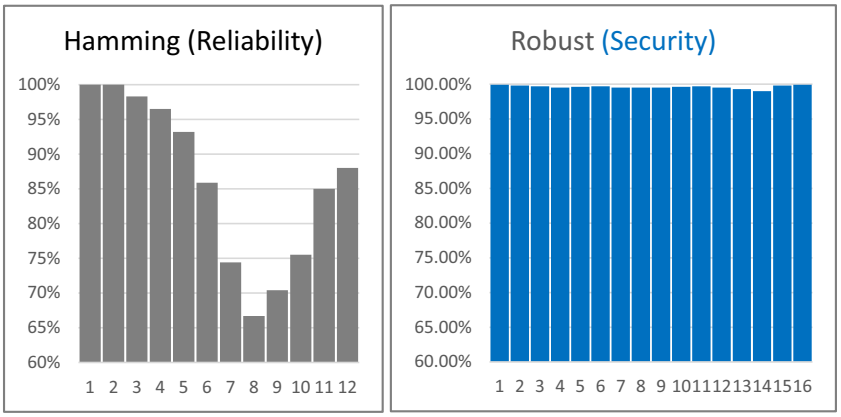


Fig. 13 The Hamming code-based SCC provides 100% of error detection on S-Box under single and double bits errors, while its performance drops drastically when the multiplicity increases. The non-linear Robust codes SCC has a high probability (> 99%) of detection of all errors

Therefore, if there exists $e = \{e_{R_b}, e_b\}$ which satisfies $H(e_{R_b}, e_b) = 0$, then we have:

$$H(\tilde{R_b}, \tilde{b}) = H(R_b, b) \oplus H(e_{R_b}, e_b) = 0 \oplus 0 = 0. \quad (16)$$

Namely, this error distorts the output of the S-box, while making itself invisible to the self-checking checker regardless of the input. For a $(12, 8, 3)_2$ Hamming code used in this example, there are totally 2^8 such invisible errors, which is the same size of this Hamming code's codeword. This type of errors can get the system stuck at state X_v from being restored to X_r or X_s.

In other words, the set size of the invisible errors to make the following equation always true is $|e_{invi}| = 2^8 - 1$ (the error of all 0 is excluded). This shows that there exists this many errors that can fail the design in Step 2 of Fig. 10 when intentional errors are injected to the Hamming code-based SCC.

$$f_{vr}(\sum l + \sum ia) = X_v \quad (17)$$

However, for the second design using a Robust code [41] based SCC, since it is non-linear, there does not exist such a relationship in Eq. 16, and so there is no error masking itself to all the S-box inputs. In other words, the set size of the invisible errors to make the following equation always true [9] is $|e_{invi}| = 0$.

$$f_{vs}(\sum l + \sum ia) = X_v \quad (18)$$

Thus, this design will pass the evaluation Steps 1, 2, and 3 in Fig. 10 since it detects both random and intentional errors with almost equal probability.

In addition, although with higher cost in hardware, the security-oriented SCC decoder using Robust codes provide better error detection probability in some multiplicities of errors as shown in the figure below (x-axis is the multiplicity of errors, and y-axis the error detection probability).

Therefore, now we can see the differences between the two protections over S-box (Fig. 13). The former guarantees up to double error detection, while its error detection probability drops drastically when the multiplicity of errors increases. The latter's detection probability stays at a satisfying leveling regardless of the number of errors. More importantly, the Hamming code-based design has a critical vulnerability of the invisible errors, which could be leveraged easily by attackers to put the system at X_v at all time, while the Robust code-based design does not (Fig. 14). Therefore, the latter should be adopted as the SCC design when there is a high demand of security in the S-BOX.

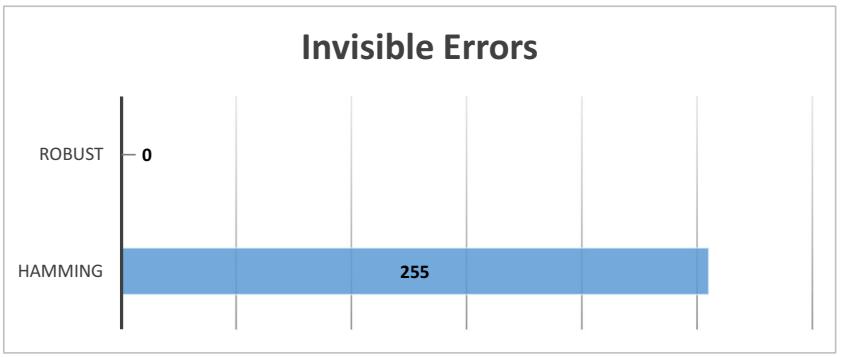


Fig. 14 Moreover, there are in total $2^8 - 1$ invisible errors in the Hamming code-based protection scheme. These errors can be easily calculated given the Hamming code's primitive polynomial, which are usually not confidential. Once any of these errors appear in the S-BOX, no matter what the input value will be, its output will always be considered as legal by the self-checking checker

Table 2 Hardware overhead comparison

Stages	P_{det} (H, %)	P_{det} (R, %)	Overhead (H, %)	Overhead 212 (R, %)
AddRoundKey	69.2	99.4	22.1	56.3
SubBytes	70.4	99.3	23.7	63.7
ShiftRows	65.3	99.6	30.5	80.3
MixColumns	68.8	99.6	35.6	77.9

[I] P_{det}: average probability of error detection under all multiplicity of errors;
[II] H: Hamming code-based scheme, and R: Robust code-based scheme

However, better trustworthiness always comes with more cost. Here, we extend the scheme to all four stages of AES, and the hardware overhead of Hamming code-based scheme and Robust code-based are compared by:

$$HardwareOverhead = \frac{\text{Stage} + \text{SCC}}{\text{Stage}} - 1$$

It can be seen from the following table that although the Robust code-based protection provides much higher error detection probability, its hardware overhead is also 2 to 3 times of the Hamming code-based version. Therefore, usually designers should also look into the performance and cost trade-off to make the proper decision (Table 2).

6.2 Case Study of Fault Attacks on SIMON

In this example, we used a lightweight cipher, SIMON [4] with a 64-bit plain text and 96-bit key as a case study to analyze the impact of different types and numbers of faults on the system states. It is not necessary to enumerate all possible system states. We used power consumption to differentiate diverse system states. In this case study, we implemented SIMON in an iterative fashion. The Verilog HDL code of SIMON was synthesized, placed, and routed in the Xilinx ISE 14.1 design suite. The power consumption was measured by the tool XPower Analyzer. We injected single-bit, 32-bit, and 64-bit faults. Three types of faults, Stuck-at-0, stuck-at-1, and Bit flip faults, are emulated for different bit width faults. All the faults were placed in the registers used by the last round state of SIMON. The power consumption after the last round computation was measured. The results of this experiment are demonstrated in Fig. 15.

As explained in the Section 5.2, if the random intrusions $\sum ia$ or $\sum ia + \sum ip$ happen then the normal state will transition to vulnerable state X_v as depicted in (19). If the system is operating normally, then the system's power consumption is 1.113 W. However, on random intrusion, the power consumption will change. For example, if the single Stuck-at-0 fault happens, then the power consumption is reduced by nearly half. The different power consumption shown in the Fig. 15 represents the different state transitions to vulnerable states X_v because of the diverse faults induced in the SIMON state register:

$$f_{vs}(\sum l + (\sum ia + \sum ip)) \rightarrow X_v \tag{19}$$

This will indicate a disqualification in the SRASA evaluation flow in Step 2 of Fig. 10, or a pitfall found out by SRASA design flow [45] that designers should avoid.

The next experiment we conducted is to detect the intrusion in SIMON using five different fault detection methods. The five fault detection methods are double modular redundancy (DMR) [13], inverse function [26], XORing two masking vectors (i.e., DuoMask), permutation (i.e., PermDeperm) [20], and combination of masking and permutation (i.e., Permutation and Masking) [14]. All these methods were implemented in gate level and the corresponding HDL code was synthesized in the Xilinx ISE 14.1 design suite. The hardware cost of intrusion detection is listed in Table 3. The second, third, and fourth columns also indicate the system enters different states when different f_{vr} functions are activated in the system. The Permutation and Masking [14] fault detection method cost 28.5%, 38.4%, and 11.9% more slice registers, slice LUTs and occupied slices respectively compared to that of the most efficient fault detection method—inverse function [26]. However, the fault bypass rate for random faults (stuck-at-0 and bit

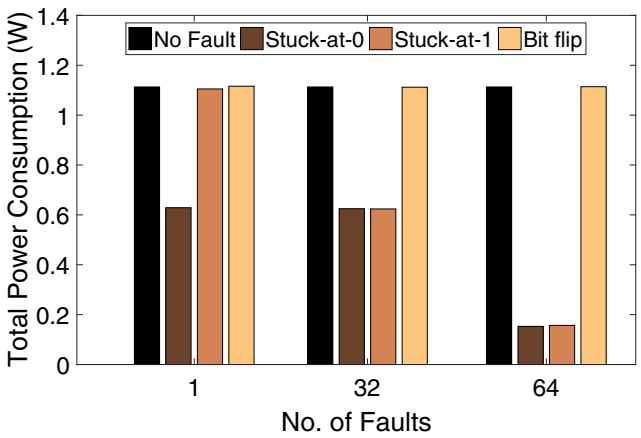


Fig. 15 Comparison of total power in different fault insertion scenarios

Table 3 Hardware cost in FPGA

Design	No. of slice registers	No. of slice LUTs	No. of occupied slices
Baseline	174	193	51
DMR	349	456	141
Inverse	175	221	96
DuoMask	180	240	95
PermDeperm	241	309	116
Permutation and masking	245	359	109

flip) and symmetric faults is much better than the inverse function [26].

6.3 Case Study of Side-Channel Attacks on AES

In this experiment, we defined the different states of the AES gate-level implementation by analyzing the correlation coefficient verses number of power traces required to retrieve the secret key. We assume that an AES hardware module is attacked by an intrusion function f_{sv}, which causes the leakage of the side-channel information—power in this case study. For the baseline AES (no protection against the power-based side-channel attacks), as shown in correlation analysis in Fig. 16a, all key bytes are recovered in 2500 power traces (red lines above the green area (wrong key guesses) show that the guessed subkeys match with the actual key) shown by the blue oval. Hence, this AES implementation is in a vulnerable state X_v. As the side-channel attack is passive, it will not cause any change in the output and it is impossible to restore the original state to X_s (secure state). We carried the similar analysis on the AES that is protected against power analysis using masking function. The results are displayed in Fig. 16, and we can notice that no Subkey is retrieved within 2500 power traces. In this scenario, the vulnerable state X_v can be restored to the secure state X_s.

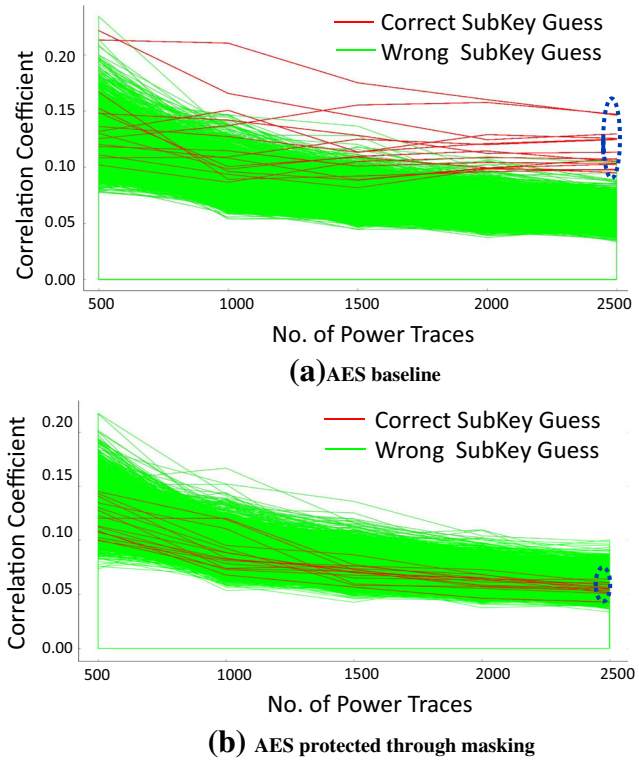


Fig. 16 Correlation coefficients for AES with or without protection against side-channel attack

7 Conclusions and Future Work

In this work, first, we provide brief overviews of security and reliability in computing systems at different abstraction levels. Second, we compare and contrast the salient features of reliability and security. Finally, we introduce a generalized theoretical framework, called security and reliability aware state automaton (SRASA), to formally reason concurrently about the security and reliability aspects of computer systems. We use three case studies to introduce and illustrate the SRASA framework, a 22-tuple finite state machine model that encompasses both physical and abstract states of the system and covers both passive and active attacks. Future work will consist of (a) robustness analysis of the SRASA framework, (b) its application to a wider range of concrete design cases, and (c) refinement of the framework. We also plan to develop an open-source software tool to automate the process of mapping a design specification and implementation to the SRASA framework.

References

1. [n. d.]. ([n. d.]). http://www.columbia.edu/cu/computinghistory/ascc.html
2. [n. d.]. ARM Security Technology – Building a Secure System using TrustZone Technology (2009) http://infocenter.arm.com/help/topic/com.arm.doc.prd29genc009492c/PRD29GENC009492C_trustzone_security_whitepaper.pdf. ([n. d.])
3. Bar-El H, Choukri H, Naccache D, Tunstall M, Whelan C (2006) The sorcerer's apprentice guide to fault attacks. Proc IEEE 94 2:370–382. https://doi.org/10.1109/JPROC.2005.862424

4. Beaulieu R, Shors D, Smith J, Treatman-Clark S, Weeks B, Wingers L (2015) The SIMON and SPECK lightweight block ciphers. In: Proceedings of the 52nd annual design automation conference (DAC '15). ACM, New York, Article 175, p 6. https://doi.org/10.1145/2744769.2747946
5. Bhattacharya S, Mukhopadhyay D (2016) Curious case of rowhammer: flipping secret exponent bits using timing analysis. Springer Berlin Heidelberg, Berlin, pp 602–624. https://doi.org/10.1007/978-3-662-53140-2_29
6. Bjorklund H, Martens W ([n. d.]). The Tractability Frontier for NFA Minimization âÉĘ([n. d.])
7. Boraten T, Kodi AK (2016) Mitigation of denial of service attack with hardware Trojans in NoC architectures. In: 2016 IEEE international parallel and distributed processing symposium (IPDPS), pp 1091–1100. https://doi.org/10.1109/IPDPS.2016.59
8. Boyko KC, Gerlach DL (1989) Time dependent dielectric breakdown at 210 Aring; oxides. In: 27th annual proceedings., International reliability physics symposium, pp 1–8. https://doi.org/10.1109/RELPHY.1989.36309
9. Bu L, Karpovsky M (2016) A hybrid self-diagnosis mechanism with defective nodes locating and attack detection for parallel computing systems. In: Proceedings of IEEE on-line testing symposium (IOLTS)
10. Cherkaoui A, Fischer V, Aubert A, Fesquet L (2013) A self-timed ring based true random number generator. In: 2013 IEEE 19th international symposium on asynchronous circuits and systems, pp 99–106. https://doi.org/10.1109/ASYNC.2013.15
11. Conti M, Dragoni N, Lesyk V (2016) A survey of man in the middle attacks. IEEE Commun Surv Tutorials 18(3):2027–2051. https://doi.org/10.1109/COMST.2016.2548426
12. Costan V, Devadas S (2016) Intel SGX explained. cryptology ePrint Archive Report 2016/086. http://eprint.iacr.org/2016/086
13. Di Natale DMRG, Doulcier M, Flottes ML, Rouzeyre B (2009) A reliable architecture for parallel implementations of the advanced encryption standard. J Electron Test 25(4):269–278. https://doi.org/10.1007/s10836-009-5106-6
14. Dofe J, Frey J, Pahlevanzadeh H, Yu Q (2015) Strengthening SIMON implementation against intelligent fault attacks. IEEE Embed Syst Lett 7(4):113–116. https://doi.org/10.1109/LES.2015.2477273
15. Fournaris AP, Fraile LP, Odysseas K (2017) Exploiting hardware vulnerabilities to attack embedded system devices: a survey of potent microarchitectural attacks. Electronics 6(3):2079–9292. https://doi.org/10.3390/electronics6030052
16. Gassend B, Clarke D, van Dijk M, Devadas S (2002) Silicon physical random functions. In: Proceedings of the 9th ACM conference on computer and communications security (CCS '02). ACM, New York, pp 148–160. https://doi.org/10.1145/586110.586132
17. Gassend B, Lim D, Clarke D, van Dijk M, Devadas S (2004) Identification and authentication of integrated circuits: research articles. Concurr Comput Pract Exper 16(11):1077–1098. https://doi.org/10.1002/cpe.v16:11
18. Grasser T, Kaczer B, Goes W, Reisinger H, Aichinger T, Hehenberger P, Wagner PJ, Schanovsky F, Franco J, Roussel P, Nelhiebel M (2010) Recent advances in understanding the bias temperature instability. In: 2010 international electron devices meeting, pp 4.4.1–4.4.4. https://doi.org/10.1109/IEDM.2010.5703295
19. Gruss D, Maurice C, Mangard S (2015) Rowhammer.js: a remote software-induced fault attack in JavaScript. CoRR arXiv:http://arxiv.org/abs/1507.06955
20. Guo X, Karri R (2013) Recomputing with permuted operands a concurrent error detection approach. IEEE Trans Comput-Aided Des Integr Circ Syst, 32. https://doi.org/10.1109/TCAD.2013.2263037
21. Herder C, Yu MD, Koushanfar F, Devadas S (2014) Physical Unclonable functions and applications: a tutorial. Proc IEEE 102:1126–1141. https://doi.org/10.1109/JPROC.2014.2320516
22. Holzer M, Kutrib M (2011) Descriptional and computational complexity of finite automata: ÂĂÂĂA survey. Inf Comput 209(3):456–470. https://doi.org/10.1016/j.ic.2010.11.013 Special Issue: 3rd International Conference on Language and Automata Theory and Applications (LATA 2009)
23. Huang P-T, Fang W-L, Wang Y-L, Hwang W (2008) Low power and reliable interconnection with self-corrected green coding scheme for network-on-chip. In: Second ACM/IEEE international symposium on Networks-on-Chip
24. Jiteurtragool N, Wannaboon C, Masayoshi T (2015) True random number generator based on compact chaotic oscillator. In: 2015 15th international symposium on communications and information technologies (ISCIT), pp 315–318. https://doi.org/10.1109/ISCIT.2015.7458370
25. Johnson Jonathan, Howes W, Wirthlin M, McMurtrey DL, Caffrey M, Graham P, Keith M (2008) Using duplication with compare for on-line error detection in FPGA-based designs. Aerospace Conference
26. Karri R, Wu K, Mishra P, Kim Y (2002) Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. IEEE Trans Comput-Aided Des Integr Circ Syst 21:1509–1517. https://doi.org/10.1109/TCAD.2002.804378
27. Kim Y, Daly R, Kim J, Fallin C, Lee JH, Lee D, Wilkerson C, Lai K, Mutlu O (2014) Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In: 2014 ACM/IEEE 41st international symposium on computer architecture (ISCA), pp 361–372. https://doi.org/10.1109/ISCA.2014.6853210
28. Kocher P (1999) The Intel Âő random number generator cryptography research, Inc. White Paper Prepared for Intel Corporation
29. Lin S, Kim Y-B, Lombardi F (2011) A 11-transistor nanoscale CMOS memory cell for hardening to soft errors. In: IEEE transactions on very large scale integration (VLSI) systems
30. Nordrum A (2016) Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated. Available at, http://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated
31. Piret G, Quisquater J-J (2003) A Differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In: International workshop on cryptographic hardware and embedded systems. Springer, Berlin
32. Qiao R, Seaborn M (2016) A new approach for rowhammer attacks. In: 2016 IEEE international symposium on hardware oriented security and trust (HOST), pp 161–166. https://doi.org/10.1109/HST.2016.7495576
33. Ravi S, Raghunathan A, Chakradhar S (2004) Tamper resistance mechanisms for secure embedded systems. In: Proceedings of the 17th international conference on VLSI design, pp 605–611. https://doi.org/10.1109/ICVD.2004.1260985
34. Razavi K, Gras B, Bosman E, Preneel B, Giuffrida C, Bos H, Shui FF (2016) Hammering a needle in the software stack. In: 25th USENIX security symposium (USENIX Security 16). USENIX Association, Austin, pp 1–18
35. Rührmair U, Xu X, Sölter J, Mahmoud A, Majzoobi M, Koushanfar F, Burleson W (2014) Efficient power and timing side channels for physical unclonable functions. Springer, Berlin, pp 476–492. https://doi.org/10.1007/978-3-662-44709-3_26
36. Seaborn M, Dullien T (2016) Exploiting the DRAM rowhammer bug to gain kernel privileges. https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf

37. Takeda E, Suzuki N (1983) An empirical model for device degradation due to hot-carrier injection. IEEE Electron Device Lett 4:111–113. https://doi.org/10.1109/EDL.1983.25667
38. Tewari A, Srivastava U, Gupta P (2002) A parallel DFA minimization algorithm. In: Sahni S, Prasanna VK, Shukla U (eds) High performance computing — HiPC 2002. Springer, Berlin, pp 34–40
39. Tu KN (2003) Recent Advances on electromigration in very-large-scale-integration of interconnects. J Appl Phys 94(9):5451–73
40. Valmari A, Lehtinen P (2008) Efficient minimization of DFAs with partial transition functions. CoRR arXiv:0802.2826.2008
41. Wang Z, Karpovsky M (2010) Robust FSMs for cryptographic devices resilient to strong fault injection attacks. In: Proceedings IEEE on-line testing symposium (IOLTS)
42. Woodruff J, Watson RNM, Chisnall D, Moore SW, Anderson J, Davis B, Laurie B, Neumann PG, Norton R, Roe M (2014) The CHERI capability model: revisiting RISC in an age of risk. In: Proceeding of the 41st annual international symposium on computer architecture (ISCA '14). IEEE Press, Piscataway, pp 457–468
43. Xiao Y, Zhang X, Zhang Y, Teodorescu R (2016) One bit flips, one cloud flops: cross-VM row hammer attacks and privilege escalation. In: 25th USENIX security symposium (USENIX Security 16). Austin, pp 19–35
44. Yu Q, Frey J (2013) Exploiting error control approaches for hardware Trojans on Network-on-Chip links. In: 2013 IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFTS), pp 266–271. https://doi.org/10.1109/DFT.2013.6653617
45. Yu Q, Zhang Z, Dofe J (2018) Investigating reliability and security of integrated circuits and systems. In: IEEE computer society annual symposium on VLSI (ISVLSI)