To verify or tolerate, that's the question

Inês Pinto Gouveia
ines.gouveia@uni.lu
University of Luxembourg, Interdisciplinary Center for Security, Reliability and Trust (SnT) - CritiX group
Esch-sur-Alzette, Luxembourg

Mouhammad Sakr
mouhammad.sakr@uni.lu
University of Luxembourg, Interdisciplinary Center for Security, Reliability and Trust (SnT) - CritiX group
Esch-sur-Alzette, Luxembourg

Rafal Graczyk
rafal.graczyk@uni.lu
University of Luxembourg, Interdisciplinary Center for Security, Reliability and Trust (SnT) - CritiX group
Esch-sur-Alzette, Luxembourg

Marcus Völp
marcus.voelp@uni.lu
University of Luxembourg, Interdisciplinary Center for Security, Reliability and Trust (SnT) - CritiX group
Esch-sur-Alzette, Luxembourg

ABSTRACT

Formal verification carries the promise of absolute correctness, guaranteed at the highest level of assurance known today. However, inherent to many verification attempts is the assumption that the underlying hardware, the code-generation toolchain and the verification tools are correct, all of the time. While this assumption creates interesting recursive verification challenges, which already have been executed successfully for all three of these elements, the coverage of this assumption remains incomplete, in particular for hardware. Accidental faults, such as single-event upsets, transistor aging and latchups keep causing hardware to behave arbitrarily in situations where such events occur and require other means (e.g., tolerance) to safely operate through them. Targeted attacks, especially physical ones, have a similar potential to cause havoc. Moreover, faults of the above kind may well manifest in such a way that their effects extend to all software layers, causing incorrect behavior, even in proven correct ones. In this position paper, we take a holistic system-architectural point of view on the role of trusted-execution environments (TEEs), their implementation complexity and the guarantees they can convey and that we want to be preserved in the presence of faults. We find that if absolute correctness should remain our visionary goal, TEEs can and should be constructed differently with tolerance embedded at the lowest levels and with verification playing an essential role. Verification should both assure the correctness of the TEE construction protocols and mechanisms as well as help protecting the applications executing inside the TEEs.

CCS CONCEPTS

• **Hardware** → **Fault tolerance**; • **Theory of computation** → **Program verification**; • **Security and privacy** → **Systems security**.

KEYWORDS

TEE, verification, resilience

This work is partially supported through the FNR Core grants HyLIT and ByzRT.

PaveTrust '21, Dec. 06, 2021, Virtual Conference
2021. ACM ISBN 978-1-4503-XXXX-X/21/12…$0.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Inês Pinto Gouveia, Mouhammad Sakr, Rafal Graczyk, and Marcus Völp. 2021. To verify or tolerate, that's the question. In *PaveTrust '21: Program Analysis and Verification on Trusted Platforms, Dec. 06, 2021, Virtual Conference.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Trusted Execution Environments (TEEs) aim at providing, inside the same computing system, a more secure environment for executing arbitrary code next to untrusted, potentially compromised, or malicious code (outside that environment).

There are many possibilities to implement such an environment. However, aside from the secure co-processor [13] approach (i.e., by providing a dedicated, isolated core as TEE-hosting unit), commercially pioneered by IBM well before TrustZone [2] and Intel SGX [10], modern TEE-enabled architectures typically aim at using the same computing elements for the TEE and for the environment in which it is embedded (e.g., by offering special processor modes for secure/non-secure world switches or for embedding trusted code into applications).

The above naturally raises the question "Why TEEs are needed in the first place?" or, more precisely, "What makes TEEs more trustworthy than the 'normal' execution environments that the platform offers?". This is in particular the case if the microcode needed to implement the TEE is of a complexity comparable to that of a small microkernel or microhypervisor. Such kernels are dedicated to isolating tasks for the purpose of confining software-level faults. As such, they fulfill as well the essential requirements commonly placed on TEEs, namely protection of the integrity and (at least to some extent) confidentiality of the code and data executed inside the TEE. In fact, Hofmann et al. [24] implemented a scheme similar to Intel SGX on top of their Inktag microhypervisor. In both, enclave resource allocation remains under control of an untrusted operating system (OS), which executes on top of this kernel / on the x86 hardware. Of course the classical argument remains valid, in that code inside a TEE becomes largely independent of complex OS services, such as network stacks, memory allocation, etc. However, this does not extend to the functionality that microkernels and microhypervisors provide. Their sole purpose is to isolate subsystems, while providing them the means to securely overcome this isolation in a controlled manner [31].

A related question along these lines is whether microcode or microkernel protected tasks, plus the generic hardware required to execute a program, is really the smallest reliable computing base (RCB) [11]. The RCB of a component is the set of those components that one must necessarily trust for the former to reliably produce correct results. As such, the RCB extends the concept of a trusted computing base from integrity and confidentiality to include also reliability and availability.

The above question becomes particularly relevant if the system faces targeted attacks or operates in environments where the accidental fault ratio is increased. Microkernel correctness, and, in fact, the correctness of executing any kind of code, hinges on the correctness of the mechanisms that the hardware platform provides for isolating user code from kernel code (privileged mode, gates, etc.) and for isolating user-level tasks one from another (e.g., memory management units (MMUs), memory protection units (MPUs) and similar protection mechanisms). Gates secure the transition between user and kernel mode by preventing the user from entering privileged mode at an arbitrary instruction pointer. MMUs implement address translation with the help of page tables mapping each virtual address v to the physical address p of a system resource (typically memory but also memory-mapped IO), and a set of access rights R, typically $\subseteq \{read, write, execute\}$. MPUs define regions with different R for the identity mapping $v = p$. On top of the above, execution units (i.e., the instruction fetch, decode, execute, writeback and control-flow pipeline stages), as well as the memory subsystem, must all work correctly to produce correct configurations. This is particularly true when interfacing with the above protection mechanisms (e.g., with the MMU through page tables). Verification of the kernel ensures that this interfacing happens correctly, but of course under the assumption that hardware failures do not cancel these guarantees.

With the exception of partially reducing the dependency on the general memory subsystem, the same applies to microcode implementations, which are commonly booted in a closed boot manner [1] during one of the early system startup phases. Microcode further has access to dedicated system resources that are not made available to the application programmer (e.g., scratchpad memory near the core).

Closed boot, and more so authenticated boot, relies on strong cryptographic primitives and on their correct implementation, likewise spawning a recursive verification challenge for which several positive examples have been published [21, 45].

Unfortunately, also this assumption of "verified programs will not be failing" remains incomplete in the presence of hardware faults, whether caused accidentally or intentionally maliciously, for example, by exploiting behavior that was not considered in the formal model (e.g., side channels).

Cryptanalysis ensures sufficient key length are used such that the residual risk of adversaries breaking the cipher through analysis remains tolerably low. A statement, which although orders of magnitude less severe than hardware faults, already alludes to "good enough", but not "perfect" systems. But cryptanalysis is only one way to break ciphers, requiring keys to be rotated and algorithms to be replaced, which might remain challenging the deeper such an algorithm is embedded into the system (e.g., for system startup) and the longer the system lives (e.g., cars, planes, trains).

[1] Closed boot restricts the software that can be started (e.g., to those carrying a specific signature). Open boot or authenticated boot allows any software to be started by measures what has been booted (e.g., by computing a secure hash of the booted code, its data and configuration parameters).

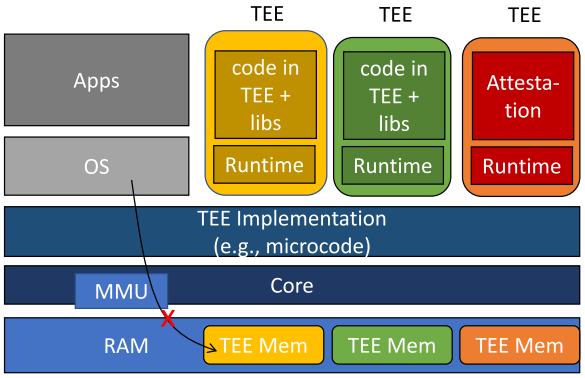

Figure 1: TEE in a typical hardware/software stack. The TEE implementing microcode (or kernel) configures the core privilege enforcement mechanism (e.g., MMU) to deny the untrusted legacy OS access to TEE memory while the TEE is active. Code inside the TEE, including its libraries and runtime environment, may fail due to software faults or faults in the TEE implementation layer or in the core.

Of course, in the above we have taken two extreme, and in fact questionable positions, namely that absolute protection is required and that formal verification is a way to get there, without properly justifying either of them. However, instead of justifying them now, we would rather like to ask the question "What would be the consequence to the latter if the first would not hold?". That is, how would verification look like if "good enough" protection, security and correctness are the goal? What is "good enough" and how can we quantify that? We hope that by now we have inspired the verification engineers in this workshop to think about possible answers to these questions. We will share our thoughts in Section 6. However, before that, let us dive a little deeper into TEEs and their reliable computing base. Section 2 details TEE implementations and analyzes them from a complexity and RCB perspective. Section 3 surveys tolerance mechanisms as a potential alternative route to reducing the RCB, in particular of the TEE itself. In Section 4, we bring both together, by hinting to incremental means of hardware-enforced resilience, discussing implications for TEE construction in Section 5 and the verification challenge this implies. We return to the previous question in Section 6 to then conclude in Section 7, by answering the main question of this paper: *to verify or to tolerate.*

2 TEES AND THEIR COMPLEXITY

From the requirement to offer a generic environment for executing arbitrary code, protected from code outside the TEE and from code in other TEEs [2], it is already possible to derive a certain minimally induced complexity.

[2] Of course one could imagine architectures with a single TEE, but TEE-to-TEE protection enables interaction and hierarchy (e.g., for the boot process).

One aspect, in which this complexity manifests, is the requirement that TEEs must necessarily be provided with memory to hold code and data and that this memory must be private. The private memory abstraction (i.e., only the TEE can read and write private locations) is typically guaranteed by the operating system kernel, managing this memory. However, contemporary architectures equate the right to manage memory (and other platform resources) with the right to obtain access to such memory. That is, privilege enforcement mechanisms (e.g., MMUs and MPUs) are constructed such that the OS has full control over which memory and access rights it grants to which application. In particular, the OS itself often exploits virtual memory, by controlling its own page tables, and already this gives the OS the possibility to insert virtual-to-physical address translations to arbitrary locations, including to the private memory of the applications it manages. In fact, such access is crucial to implement OS functionality such as transparent compression, swapping, and migration (e.g., of virtual machines). Even if confidentiality is less of a concern, the integrity of applications depends on them being the sole writer of their data and on no other entity manipulating their code.

The above requirement either automatically drags the OS into the reliable computing base (RCB) of all applications or requires other means to deprivilege the managing instance, while protecting the managed execution environment. Microkernel-based system construction advocates splitting OS functionality into dedicated user-level servers to limit the complexity of an application to only include those servers that it uses (and the transitive closure of this "use-therefore-trust" relationship). The Inktag microhypervisor and Intel SGX for memory, and Hohmuth et al. [25] and Weinhold et al. [48] for higher-level objects, break this transitivity by crytographically protecting integrity and confidentiality before data is exposed to the resource managing OS (e.g., the Linux file system, which remains responsible for storing encrypted files to disk). The microkernel and large parts of the hardware stack remain part of the RCB and, in fact, the data storing Linux as well, as far as availability of the data is concerned.

Cryptography also plays an essential role in authenticating and attesting TEEs and the code they execute for the purpose of establishing initial trust in the functionality they offer. Attestation further unlocks a deployed application's sealed memory [14], that is, private storage that is accessible only by authenticated applications and that can be used, for example, to store further key material for the application.

There are two complexities hidden in authentication and attestation (aside of course the need for enough computation power to compute the algorithms):

(1) the requirement to support multiple TEEs, protected from each other, to convert the secure boot of the attestation enclave into an authenticated boot infrastructure of late-launched enclaves; and
(2) the need to frequently replace key material and possibly even the crypto algorithm itself.

The first is not inherent, but primarily an artifact of current attestation protocols, independently reporting quotes for booted applications. As an alternative, one could equip applications with a proof that they have been booted in an authenticated manner, as exemplified in Gross [23] [3]. In a nutshell, the approach by Gross generates a random key that the booted subsystem must keep in volatile memory and certifies that the key holder has been authenticated. This way, signatures with this key serve as proof of possessing a secret, which, due to the volatile storage requirement, is only valid until the next reboot, after which a new key is generated.

Let us here only discuss one further complexity, which originates from the functionality that programs expect from the environment they execute in. Complex applications inside the TEE require the same set of libraries, resources, etc., that they would require outside the trusted execution environment. Provisioning such functionality inside the TEE quickly reaches a point where the system deployed inside is only marginally less complex than the system outside. For example, preemptive multithreading inside a single TEE requires the very same context-switching code that an OS has to deploy for switching between multiple application-level threads.

The complexity just mentioned is inherent in classical settings and can only be mitigated by disciplining one self to limit what should go into a TEE.

Fortunately, in addition to the vertical [4] trust relationships just discussed, there is also the possibility to break transitive trust relationships horizontally. Rather than universally trusting a component a subsystem uses, one could aim for trusting a healthy majority of such components, operating in consensus. This pattern is commonly found in fault tolerant settings, which we discuss next before we return to the question how the reliable computing base one inherts from using a TEE can be reduced. A question that becomes particularly relevant for those applications of TEEs that offload only simple functionality into these environments.

3 FROM TOLERANCE TO RESILIENCE

Fault tolerance schemes such as triple-modular redundancy (TMR) [37] and byzantine fault tolerant state-machine replication (BFT-SMR) [7, 29] instantiate the same functionality (though not necessarily the same code) across multiple replicas and require a certain threshold of them to agree on their outputs, respectively on the operations they perform. A key assumption here is that the same fault will not harm more than f replicas simultaneously, such that a healthy majority of $n - f$ replicas remain to mask faulty behavior. Typically $n = 2f + 1$ for TMR-like or hybrid BFT-SMR schemes [8, 30, 45], where equivocation is prevented either through the invocation pattern or by inclusion of a special component that is trusted to fail only by crashing. $n = 3f + 1$ if replicas have to prepare for faulty peers lying inconsistently.

Fault tolerance schemes are prepared to cope with faults of an accidental nature (such as radiation-induced single event upsets), but they can also be applied to fend off adversaries, when combined with rejuvenation [41, 42] to return compromised replicas to a state at least as secure as initially, and diversification [18, 19, 36] to present adversaries a moving target, cancelling knowledge how replicas can be attacked.

[3] See Appendix A for an english summary of the proposed boot algorithm and its extension to sealed memory.
[4] We use the term *vertical* following the commonly used depiction of software stacks (e.g., in Figure 1) where functionality users stack on top of functionality providers.

Replication can even help protecting secrets, if threshold cryptography is applied to share secrets such that revealing up to f of its parts conveys no information [51].

Traditionally replication schemes are applied in a distributed systems context, with protocols such as crash-fault tolerant Paxos [28], Byzantine fault tolerant PBFT [7] and Hotstuff [50], and hybrid protocols such as MinBFT [45], possibly even leveraging TEEs to implement the trusted component they need [20], even deploying verified code in such environments [46]. However, Paxos and BFT replication have as well been attempted inside multiprocessor systems on a chip (MPSoCs) [4, 6, 11, 16, 28], under the assumption that the low-level kernel (e.g., hypervisor or platform manager) is trustworthy. Obviously, such a kernel would be a single point of failure (SPoF) and one of a non-trivial complexity. But is the processor and a somewhat small kernel (microcode or software) inherently part of all RCBs or can we do without?

4 HARDWARE ENFORCED RESILIENCE

Clearly homogeneous approaches will not work as long as we aim to retain the flexibility MPSoCs offer, so lets aim at heterogeneous settings, where some functionality must not fail. *Homogeneous* and *heterogeneous* here pertain to the fault model. In the former, all components are treated alike. They may fail in the way considered by the fault model (e.g., by crashing or, worse, by exhibiting arbitrary and potentially malicious, Byzantine behavior, for example, after they have been compromised by an adversary who is now in control of such components).

Heterogeneous fault models distinguish how some selected components may fail. For example, while the majority of all components, in particular the complex ones, may fail in an arbitrary, possibly Byzantine manner, small, low-complex, trusted components are assumed to fail only by crashing and possibly in a detectable manner or not at all. As long as this is the case, the high-complex components can benefit from the latter to secure certain "good" behavior despite faults. For example, MinBFT's USIG [45] prevents replicas from creating different answers to the same request, by attaching, in a trusted manner, a unique sequential identifier, which the USIG protects cryptographically.

Transactional schemes have been proposed to protect state from transient errors, by preparing for an automatic restart [34] or by cross-checking transactional state (e.g., with checksums) [3]. Already the early IBM 370 had RAS units to compare pipeline results against a redundant copy and lockstep execution is a widely deployed scheme in safety-critical systems. The challenge remains so, since the stronger such schemes force code to behave identically, the weaker their protection against a targeted attack, in particular if it exploits a vulnerability in such code, which verification has ideally found during the verification process.

Still, approaches like the above keep caches and coherence logics, that is large parts of a core's memory subsystem, in the RCB and the complex protocols that govern them. Moreover, the above approaches merely detect faults and rely on them being transient when re-executing the faulty segment or on a trusted OS for different recovery strategies. Again, the question arises whether the OS and its recovery strategies are inherently part of all RCBs or whether we can do better. It turns out we can, provided hardware developers eliminate the remaining common mode faults and if we exclude proximity faults from our fault model.

5 TOWARDS RESILIENT TEES

With Midir [22], Gouveia et al. entered a research direction where single-points of failure (SPoF), in particular at hypervisor level, are systematically avoided by subjecting all critical operations to consensus votes among a fault-threshold exceeding quorum of (kernel-)replicas. In particular, an architecture was proposed where the privilege to manage a resource does not automatically allow the manager to obtain access to the managed resource. Instead, privilege change must happen consensually through voted operations and enforced in trusted components, with the consequence that no faulty replica can perform such an operation unless it is as well authorized by a healthy majority. System calls are voted upon as well as all critical operations invoked during their execution and all that is achieved by merely two trusted functionalities: trusted voters and trusted privilege enforcement. A third one, reliable communication among these trusted components, can be lifted through proper on-chip network coding [33] and adaptive routing techniques [49].

The above obviously also comes with a set of assumptions, namely that tiles (e.g., cores and co-located memories) fail independently due to accidental causes (e.g., by powering cores from redundant supplies and by making the clock network resilient).

The downside of all tolerance approaches is of course that they require space. Fig. 2 illustrates the Midir architecture, its replicated hypervisor (TEE Impl.) and two possibilities to realize TEEs, relative to the resilience needed from the code deployed in them. Leveraging privilege reversion, hypervisor replicas agree to equip enclaves with the resources an untrusted resource manager (here Linux - Apps + Lib + OS) selects for them. However, because no healthy replica will agree to hypervisor replicas granting themselves or untrusted components access to the same memory pages, private memory is guaranteed as well relative to all hypervisor replicas. The same hardware-accelerated voting mechanism that hypervisor replicas use to agree on critical operations in system calls (such as granting privileges), can be used by code inside the TEE to agree with their peers on the results of extremely critical functionality. TEEs are therefore not only trusted to isolate applications and protect their integrity and confidentiality, they are also trustworthy by recursively basing this trust on tolerance, both at hypervisor and at TEE-deployed code level. Aside from Midir's trusted voting and privilege enforcement units, no single component is necessarily part of an application's RCB. Instead, RCBs include healthy majorities of replicated components that operate in consensus to mask faulty behavior. But how secure can such a system be? The answer to this question is rooted in the inherent resilience of each of the components in the interacting replica groups. If they are too vulnerable, more components will fail than can be tolerated; if they are too weak, adversaries will spot inherent vulnerabilities despite diversification, which allows them to outpace rejuvenation. And, last but not least, BFT-SMR, even for classical client-server patterns, is far from easy to develop, yet to get right. This is where formal verification returns into the big picture.

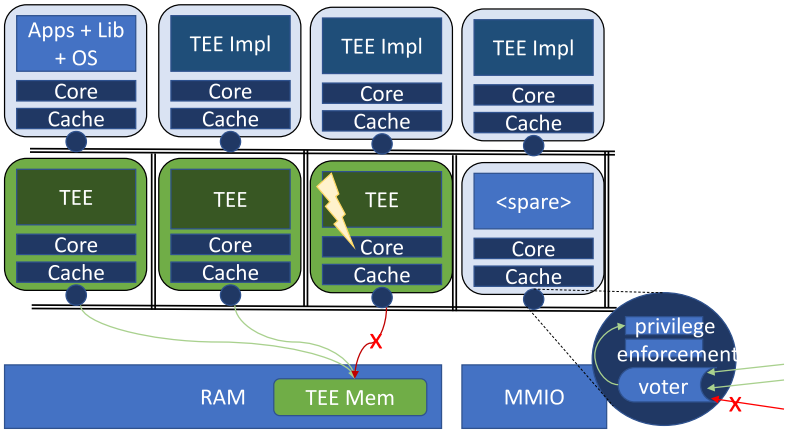

Figure 2: Trusted components, comprised of voter and privilege enforcement unit (e.g., capabilities) confine faults to cores and the resources they are authorized to access. Consensual privilege change avoids SPoF syndromes in the TEE implementation layer (e.g., a hypervisor). TEEs are just normal cores repurposed for hosting TEE code and data, as long as they are needed. The TEE implementation layer and the TEE itself can be configured to operate in a replicated manner, masking diverging minority behavior during consensual resource access, including during privilege change.

6 VERIFYING IMPERFECT CORRECTNESS

Minimal root of trust TEEs, be them constructed with Midir or otherwise, offer isolated environments that may still fail. Combining them in a replicated manner allows masking these faults under the assumption that each replica is sufficiently diverse and generally good enough to not fail under the same conditions that its peers are failing, in particular while the system is under attack. Ensuring that replicas are sufficiently diverse and generally good enough to withstand attacks is, naturally, a challenge worth exploring formally, as is the task of verifying the tolerance scheme itself.

First promising approaches, in particular towards the latter, are already in place. In [39] they devise formal models for different features of SGX, then they introduced a verification methodology to prove that an enclave program running on SGX does not reveal sensitive information to an adversary. In [44], the authors presented a methodology to design a TEE in a way that the verification of security properties is possible, and [38] introduces a methodology to design applications in a way that enables certifying their confidentiality when running inside isolated execution. Furthermore, ByMC [27] explores model checking of threshold-guarded distributed algorithms in the presence of Byzantine faults and Vukotic et al. [47] proved the seminal protocols SM, PBFT and MinBFT safe in a knowledge-based interactive theorem-proving framework, while generating Intel SGX code for MinBFT's trusted component. Let us leave no doubt that these achievements are impressive, or, to say it with the words of Zachary Tatlock during the 2018 DeepSpec Summer School: "This paper did something that actually I thought was impossible.". But what did these verification results achieve exactly? They tell us that in the presence of $n - f$ healthy replicas, performing the desired operation and executing the verified protocol, clients will always receive correct results, even if up to f replicas behave arbitrarily malicious. They do not tell us that the replicas are of a sufficient quality such that at most f fail simultaneously and they also do not tell us what this quality might be. It is true that we no longer need to worry about those that fail, provided they are at most f, but verification can do so much more, at least we hope.

There are plenty of formal verification works [1, 5, 12, 15, 32, 35] that relax the specifications to be checked in case the more general ones are "hard" to check. This relaxation is needed in case the problem is undecidable, the verification complexity is too high or if the state space explodes, even after applying all possible reduction techniques. For example, rather than verifying liveness in the strong sense — good states are reachable infinitely often ($\Box\Diamond p$) — sometimes it is only possible to show they are reached, at least once ($\Diamond p$)[1]. But is this not enough if some, ideally well below f, replicas just reach good states only once? For instance, Jacobs et al. [26] have shown that the parameterized model checking for simultaneous and repeated reachability (all processes reach some local state simultaneously) is decidable. However, this result was for a very restricted class of concurrent systems where they assume that all processes are not malicious. Furthermore, sometimes verification only remains feasible up to certain bounds on the lengths of paths [5] (i.e., up to a certain number of steps into the future) or for systems of a given dimension. Is a verification result for some bound b good enough? When is a verification result "good enough" and what are the implications to verification itself, when "good" and not "perfect" is enough?

We do not yet have definitive answers to the above questions, in particular because we do not even have a definitive answer on how to quantify resilience other than in retrospect by producing statistics on Common Vulnerabilities and Exposures (CVE) reports. Even these statistics are questionable when it comes to their expressiveness in predicting the likelihood of successful future attacks. Can we model adversaries, their incentives and when and how they may act? And if so, why should adversaries follow our models?

The story is of course slightly different, if we limit ourselves to the case of accidental faults for which statistics are generally known. Modeling, for each correct path, also the possibility of deviating

form this path in case accidental faults manifest themselves, and limiting the extend to which this deviation may happen (e.g., to single bit flips) it is possible to assess the quality of a program in the presence of accidental faults by the effects such bit flips have on the reachability of correct results. The question therefore remains how to extend this to the general case of multiple bit flips or worse, adversaries influencing the state of a program more significantly.

Another dimension of course lies in the level of abstraction [9, 17] at which the verification task is performed. Capturing only the essence of an algorithm, abstract and course grain models reveal high level properties that may well be verified at this level of abstraction, but that may not be preserved during verifiable refinements. Is the high-level verification result useless because of that? Here, we can finally give a definite answer: it is not. Remember the condition for tolerance to fend off adversaries. Replicas must present themselves to adversaries in such a way, that the time required to compromise more than f of them, does not fall below a certain minimal bound. With such a bound T_{attack}, rejuvenating all replicas faster than T_{attack} always presents the adversary a situation where it has to perform at least that amount of work before it would be able to exhaust the tolerance threshold of the system [40, 43]. Even though not all refinements would preserve the high-level result, generating enough diverse instances fast enough, we have a realistic chance of reliable safety and security without running into the risk that eventually the adversary would find a systematic flaw that persists in all such instances.

7 CONCLUSIONS

"Verify or tolerate?" Although many questions remain open — both for system researchers and engineers in how we believe TEEs should be constructed, how the inherent complexities of their construction should be avoided and how we can use them; and for verification researchers and engineers in how to assert the quality of verification in the presence of faults — we hope the answer to the above question remain "both". Verify to ensure that despite imperfection replicas remain good enough to not fall simultaneously *and* tolerate to catch for the most critical applications large parts of the residual risk of good enough components to fail in the presence of accidental and malicious faults. We have sketched how TEEs might be constructed to tolerate faults, both in the entities required to construct them and, incrementally, in the software components they host. We have also sketched how verification (in combination with diversification) help us obtain good enough replicas. However, quantifying "good enough" remains a research challenge for the future.

REFERENCES

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. 1996. General decidability theorems for infinite-state systems. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 313–321.

[2] Tiago Alves and Don Felton. 2004. TrustZone: Integrated hardware and software security. *ARM white paper* 3, 4 (2004), 18–24.

[3] Rico Amslinger, Sebastian Weis, Christian Piatka, Florian Haas, and Theo Ungerer. 2018. Redundant Execution on Heterogeneous Multi-cores Utilizing Transactional Memory. In *Architecture of Computing Systems – ARCS 2018*. Springer International Publishing, Cham, 155–167.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[5] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. (2003).

[6] Thomas C. Bressoud and Fred B. Schneider. 1995. Hypervisor-based fault tolerance. In *15th ACM Symposium on Operating Systems Principles (SOSP)*. Copper Mountain, Colorado, USA, 1–11.

[7] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[8] B. Chun, P. Maniatis andS. Shenker, and J. Kubiatowicz. 2007. Attested append-only memory: Making adversaries stick to their word. In *21st ACM Symposium on Operating Systems Principles (SOSP)*. Stevenson, Washington, USA, 189–204.

[9] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer, 154–169.

[10] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained.* Technical Report. Massachusetts Institute of Technology. https://eprint.iacr.org/2016/086.pdf (Accessed: 2016-07-22).

[11] Björn Döbel. 2014. *Operating System Support for Redundant Multithreading*. Ph. D. Dissertation. Technische Universität Dresden, Dresden, Germany.

[12] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (2008), 1165–1178.

[13] J.G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S.W. Smith. 2001. Building the IBM 4758 secure coprocessor. *Computer* 34, 10 (2001), 57–66. https://doi.org/10.1109/2.955100

[14] Paul England. 2008. Practical Techniques for Operating System Attestation. In *Trusted Computing - Challenges and Applications*, Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–13.

[15] Javier Esparza, Alain Finkel, and Richard Mayr. 1999. On the verification of broadcast protocols. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*. IEEE, 352–359.

[16] Emanuele Giuseppe Esposito, Paulo Coelho, and Fernando Pedone. 2018. Kernel Paxos. In *37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE.

[17] Cormac Flanagan and Shaz Qadeer. 2002. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 191–202.

[18] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. 2011. OS diversity for intrusion tolerance: Myth or reality?. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 383–394.

[19] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. 2014. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience* 44, 6 (2014), 735–770.

[20] Jian Liu Nadarajah Asokan Ghassan Karame, Wenting Li. 2020. METHOD AND SYSTEM FOR BYZANTINE FAULT - TOLERANCE REPLICATING OF DATA. US Patent Application: US 2020/0389310 A1.

[21] David Gilhooley. 2017. Secure Boot: Formal Verification of Software & Hardware in a large SoC. (2017).

[22] Inês Pinto Gouveia, Marcus Völp, and Paulo Esteves-Verissimo. 2020. Beyond the last line of defense - Surviving SoC faults and Intrusions. In *arXiv:2005.04096*. IFIP/IEEE.

[23] Michael Groß. 1991. Vertrauenswürdiges Booten als Grundlage authentischer Basissysteme. In *VIS '91 Verläßliche Informationssysteme*, Andreas Pfitzmann and Eckart Raubold (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–207.

[24] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: Secure Applications on an Untrusted Operating System. *SIGPLAN Not.* 48, 4 (March 2013), 265–278. https://doi.org/10.1145/2499368.2451146

[25] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. 2004. Reducing TCB Size by Using Untrusted Components: Small Kernels versus Virtual-Machine Monitors. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop* (Leuven, Belgium) *(EW 11)*. Association for Computing Machinery, New York, NY, USA, 22–es. https://doi.org/10.1145/1133572.1133615

[26] Swen Jacobs and Mouhammad Sakr. 2018. Analyzing Guarded Protocols: Better Cutoffs, More Systems, More Expressivity.

[27] Igor Konnov and Josef Widder. 2018. ByMC: Byzantine model checker. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 327–342.

[28] Leslie Lamport. 1998. The part-time parliament. *Transactions on Computer Systems* 16, 2 (1998), 133–169.

[29] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401. https://doi.org/10.1145/357172.357176

[30] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems., Vol. 9. Boston, Massachusetts, USA, 1–14.

[31] J. Liedtke. 1996. Toward Real Microkernels. *Commun. ACM* 39, 9 (1996), 70–77. https://doi.org/10.1145/234215.234473
[32] Fuchun J Lin, PM Chu, and Ming T Liu. 1987. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. In *Proceedings of the ACM workshop on Frontiers in computer communications technology*. 126–135.
[33] Simon Ogg, Bashir Al-Hashimi, and Alex Yakovlev. 2008. Asynchronous Transient Resilient Links for NoC. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (Atlanta, GA, USA) *(CODES+ISSS '08)*. ACM, New York, NY, USA, 209–214. https://doi.org/10.1145/1450135.1450182
[34] Dimitra Papagiannopoulou, Andrea Marongiu, Tali Moreshet, Luca Benini, Maurice Herlihy, and R. Iris Bahar. 2015. Playing with Fire: Transactional Memory Revisited for Error-Resilient and Energy-Efficient MPSoC Execution. *Proceedings of the 25th edition on Great Lakes Symposium on VLSI* (2015).
[35] Ganapathy Parthasarathy, Madhu K Iyer, K-T Cheng, and L-C Wang. 2004. Safety property verification using sequential SAT and bounded model checking. *IEEE Design & Test of Computers* 21, 2 (2004), 132–143.
[36] Tom Roeder and Fred B Schneider. 2010. Proactive obfuscation. *ACM Transactions on Computer Systems (TOCS)* 28, 2 (July 2010), 1–54.
[37] Martin Shooman. 2002. *N-Modular Redundancy*. 145–201. https://doi.org/10.1002/047122460X.ch4
[38] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P Lopes, Sriram Rajamani, Sanjit A Seshia, and Kapil Vaswani. 2016. A design and verification methodology for secure isolated regions. *ACM SIGPLAN Notices* 51, 6 (2016), 665–681.
[39] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2015. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1169–1184.
[40] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. 2009. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems* 21, 4 (2009), 452–465.
[41] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. 2010. Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery. *IEEE Trans. Parallel Distrib. Syst.* 21, 4 (April 2010), 452–465. https://doi.org/10.1109/TPDS.2009.83
[42] Paulo Sousa, Nuno Ferreira Neves, and Paulo Veríssimo. 2006. Proactive Resilience through Architectural Hybridization. In *Proceedings of the 2006 ACM Symposium on Applied Computing* (Dijon, France) *(SAC '06)*. Association for Computing Machinery, New York, NY, USA, 686–690. https://doi.org/10.1145/1141277.1141435
[43] Paulo Sousa, Nuno Ferreira Neves, and Paulo Verissimo. 2006. Proactive resilience through architectural hybridization. In *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 686–690.
[44] Haiyong Sun and Hang Lei. 2020. A design and verification methodology for a trustzone trusted execution environment. *IEEE Access* 8 (2020), 33870–33883.
[45] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient byzantine fault-tolerance. *IEEE Trans. Comput.* 62, 1 (2013), 16–30.
[46] Ivana Vukotic. 2020. *Formal Framework for Verifying Implementations of Byzantine Fault-Tolerant Protocols Under Various Models*. Ph. D. Dissertation. University of Luxembourg. avail at. http://hdl.handle.net/10993/43529.
[47] Ivana Vukotic, Vincent Rahli, and Paulo Esteves-Veríssimo. 2019. Asphalion: trustworthy shielding against Byzantine faults. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–32.
[48] Carsten Weinhold and Hermann Härtig. 2008. VPFS: Building a Virtual Private File System with a Small Trusted Computing Base. *SIGOPS Oper. Syst. Rev.* 42, 4 (April 2008), 81–93. https://doi.org/10.1145/1357010.1352602
[49] Pengfei Yang, Quan Wang, Wei Li, Zhibin Yu, and Hongwei Ye. 2016. A Fault Tolerance NoC Topology and Adaptive Routing Algorithm. In *2016 13th International Conference on Embedded Software and Systems (ICESS)*. 42–47. https://doi.org/10.1109/ICESS.2016.20
[50] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591
[51] L. Zhou, F. B. Schneider, and R. V. Renesse. 2005. APSS: proactive secret sharing in asynchronous systems. *ACM Transactions on Information and System Security* 8, 3 (2005), 259–286.

A GROSS' AUTHENTICATED BOOT AND ITS SEALED MEMORY

The following is a brief summary of Gross [23], which unfortunately is only available in German. In 1991, Gross proposed an authenticated boot algorithm that, unlike TPMs is based on key generation each time the system is booted.

Starting, the processor computes a hash of the first software (typically BIOS) it boots, while generating a key pair $bios\_running^{priv,pub}$, which it stores in volatile memory that is known to forget its content upon reboot. It then certifies:

$$CERT = \langle hash(BIOS), ID_{CPU}, bios\_running^{pub} \rangle \sigma_{CPU^{priv}} \quad (1)$$

The BIOS and in turn any booted layer that obtained exclusive control over the processor performs the same algorithm for the respective next layer. For example,

$$\langle hash(OS), ID_{BIOS}, OS\_running^{pub}, CERT \rangle \sigma_{bios\_running^{priv}} \quad (2)$$

while embedding as $CERT$ the key certificate received from the lower layer and while deleting $bios\_running^{priv}$ before passing control to the OS.

Being asked to authenticate (passing a random nonce for freshness), the booted system responds with its identifier, signing the nonce just received using $OS\_running^{priv}$ and providing the necessary certificates for that.

$$\langle ID_{OS}, nonce \rangle \sigma_{OS\_running^{priv}} \quad (3)$$

All certificates are rooted in the certificate of the key CPU^{pub} whose private counterpart is embedded into the CPU.

Key to the security of this approach is the fact that trusted systems will never reveal private keys and will destroy them during reboot or before passing control to the respective next layer.

Gross approach extends well to sealed memory, by unsealing a booted instance's memory before passing control to it, while keeping the sealing key in the own sealed memory.

For example, in addition to attesting the BIOS, the CPU decrypts the sealed memory of this BIOS instance into volatile memory, using CPU^{priv}. BIOS may update sealed memory, by overwriting the durably stored instance after encrypting with CPU^{pub}. In particular BIOS sealed-memory should contain a key $bios\_sm^{priv}$ used to unseal the sealed memory of the OS. The above obviously does not yet guarantee advanced features, such as sealed memory migration of freshness.