Assurance Argument Elements for Off-the-Shelf, Complex Computational Hardware

Rob Ashmore(✉) and James Sharp

Defence Science and Technology Laboratory, Fareham, Hants PO17 6AD, UK
{rdashmore,jsharp1}@dstl.gov.uk

Abstract. There are many aspects to the safe use of artificial intelligence. To date, comparatively little attention has been given to the specialist computational hardware that is used, especially within embedded systems. Consequently, there is a need to identify evidence that would support a compelling assurance argument for the safe use of off-the-shelf, large scale, complex system-on-chip designs. To that end, we summarise issues related to the use of multi-core processors in aviation, which contextualises our problem. We also discuss a collection of considerations that provide evidence to support a compelling assurance argument.

Keywords: Artificial intelligence · Computational hardware · Machine learning · Safety · Security

1 Introduction

Artificial Intelligence (AI), especially that enabled by Machine Learning (ML), is being used in an increasing number of applications, some with potential safety impacts. This has motivated a large body of work aimed at specific aspects of AI, including: susceptibility to adversarial examples [21]; explainability [19]; formal specification [16]; through-lifecycle assurance [4]; and ML safety engineering [24]. Less attention has been paid to the assurance of the computational hardware that supports AI. Correct functioning of this hardware is necessary for the correct functioning of AI. This hardware is also significantly more complex than anything that is currently being used in other safety-critical domains (e.g. aviation).

We briefly discuss (in Sect. 2) the introduction of Multi-Core Processors (MCPs) into the aviation domain, which provides helpful context. We then (in Sect. 3) present a structured set of evidence-generating activities that support the hardware-related portion of an assurance argument. Since it represents the most common case, we focus on the use of large-scale, complex, off-the-shelf hardware, in the form of System-on-Chips (SoCs). For brevity, we refer to this as AI-enabling hardware. A sumary is provided (in Sect. 4).

A. Casimiro et al. (Eds.): SAFECOMP 2020, LNCS 12234, pp. 260–269, 2020.
https://doi.org/10.1007/978-3-030-54549-9_17

2 Multi-core Processors in Aviation

Before discussing modern AI-enabling hardware it is informative to review the introduction of MCPs in the aviation domain.

Use of MCPs is beneficial because they provide increased computational power. It is necessary because many single-core processors are becoming obsolete. However, the introduction of MCPs has faced challenges and proceeded at a controlled, relatively slow pace. A review of relevant guidance material suggests that the two main challenges were interference and non-determinism [9]. Interference happens when one application unintentionally (and, typically, adversely) affects the behaviour of another. Use of shared resources, for example, interconnects and lower-level caches is a typical cause. Unless it is understood and mitigated, interference will make behaviour non-deterministic.

There are four main aspects to assuring the use of MCPs in aviation:

- Control the configuration of the MCP;
- Understand and mitigate potential interference paths;
- Verify behaviour of software applications running on the MCP;
- Implement architectural protections (beyond the level of the MCP).

3 AI-Enabling System-on-Chip

Approaches that support use of MCPs are insufficient to support the assurance of AI-enabling hardware. There are two main reasons for this. Firstly, SoCs are significantly more complex than MCPs. Secondly, aviation-related MCP guidance takes a traditional approach to safety, protecting against random chance events. We adopt a wider view that incorporates both safety and security.

We follow the aviation domain guidance in asserting that a SoC should be assured in the context of the software associated with a particular deployment. Providing generally-applicable assurance that a SoC can be used for all possible software applications is practically impossible.

The following subsections discuss topics that would be expected to contribute to an assurance argument for AI-enabling hardware. The subsections follow the four main aspects of MCP use in the aviation domain, as shown in Table 1; italics show cases where the subsection is of indirect relevance to the aspect.

Due to space limitations, the subsections do not cover all aspects of hardware assurance. Our focus on off-the-shelf, physical items means that considerations applied during design and manufacturing considerations are outside our scope.

3.1 Configuration

Configuration settings can significantly change hardware behaviour, for example, by controlling cache allocations and changing power management strategies. Consequently, it is important that configuration settings are documented, justified, controlled and monitored (including at run time).

Table 1. Considerations and relationships to main aspects of MCP use in the aviation domain (*italic* text shows an indirect relationship).

Aspect of MCP use	Consideration (Subsection)
Control configuration	Configuration (3.1)
	Host and Target (3.3)
Manage interference paths	Interference paths (3.2)
	Worst-Case Execution Time (3.5)
Verify behaviour	Host and Target (3.3)
	Test Coverage (3.4)
	Worst-Case Execution Time (3.5)
	Logging (3.6)
	Safety Islands (3.7)
	Hardware-Based Trojans (3.8)
Architectural protections	System-Level Architecture (3.9)

Microcode updates, can significantly change detailed behavioural aspects, for example, speculative execution [22]. Consequently, management of configuration settings needs to include management of microcode updates.

Particular attention also needs to be paid to debug features. These features are not intended to be used in operational settings, so they may be developed with less rigour. Debug features also provide access to low-level information, which should not generally be exposed during operational use.

3.2 Interference Paths

Initially, potential interference paths should be identified from a theoretical perspective. Buses, caches and interconnects are obvious candidates; interrupt handling routines may also be relevant. This theoretical investigation should inform an empirical investigation, which relies on software. Applications intended to be run on the system should be considered, as well as "enemy processes", which are deliberately designed to try and cause interference [13].

Much of the literature related to interference paths considers multiple software applications interfering with each other, with effects on timing. AI-enabling hardware may introduce the possibility of "self-interference", with an effect on computation results. For example, the order in which floating point numbers are summed can change the result of the summation. Unless care is taken, this effect could occur in parallelised calculations [23].

3.3 Host and Target

AI is often developed on "host" hardware, before deployment on to "target" hardware, embedded within the operational system. Differences between host and target hardware can lead to unexpected effects.

We note that some AI-enabling hardware may be suitable for developing neural networks. This may allow the same hardware to be used for development and operation. Provided the same configuration settings were used in both settings, this would largely mitigate the concerns discussed in this subsection.

Numerical precision may differ between host and target (e.g. to allow deployment on constrained hardware [12]). Exhaustively running all samples (from the Training, Test and Verification (TTV) data sets) through operational software on the target hardware protects against consequences of this difference. Whilst conceptually simple, this may be impractical. If so, a sampling-based approach may be used, provided samples are chosen with care. Useful inspiration may be drawn from approaches used to design computer-based experiments [18].

The approaches discussed above are focused on the performance of the final ML-trained model on the target hardware. From an engineering perspective, this can lead to increased risk late in a project, with host-target differences only becoming apparent towards the end of the activity. To reduce this risk there may be benefit in using simple models, that are quick to develop and analyse, to identify key differences between host and target hardware.

Large-scale, on-chip integration can increase the importance of Process Variation (PV), which occurs as a result of manufacturing imperfections. These imperfections can lead to significant variations in power consumption and timing violations [17]. Consequently, there is value in supplementing the generally-applicable testing outlined above with some level of test on each and every SoC.

3.4 Test Coverage

A complete discussion of software assurance issues is outside the scope of this paper, but a brief discussion of test coverage is relevant. Here, we follow [4], which also provides an overview of general ML assurance issues. When considering test coverage, it is helpful to consider four different domains, or sets of inputs:

1. The *input domain* space, $\mathcal{I}$, which is the set of inputs that the model can accept.
2. The *operational domain* space, $\mathcal{O} \subset \mathcal{I}$, which is the set of inputs that the model may be expected to receive when used operationally.
3. The *failure domain* space, $\mathcal{F} \subset \mathcal{I}$, which is the set of inputs the model may receive if there are failures elsewhere in the system.
4. The *adversarial domain* space, $\mathcal{A} \subset \mathcal{I}$, which is the set of inputs the model may receive if it is being attacked by an adversary.

Separate coverage arguments should be provided for all of these domains. The argument relating to $\mathcal{I}$ should consider the number of inputs tested and their distribution across the input space (potentially informed by designs for computer experiments [18]); situation coverage [2] may be useful when thinking about $\mathcal{O}$; coverage of $\mathcal{F}$ should be informed by analysis of system architecture, focusing on subsystems that acquire data that is subsequently used as an input to an ML model; and coverage of $\mathcal{A}$ should be based on an understanding of possible attacks [10].

3.5 Worst-Case Execution Time

One reason MCPs are of concern in aviation is their potential effect on Worst-Case Execution Time (WCET). This could be important if an AI-enabling SoC is being used as part of a vehicle control loop, for example. Activities discussed above, notably understanding interference paths and test coverage, should provide helpful information to support assurances related to WCET.

In addition, particular inputs may affect execution time. For example, it is well-known that poorly-implemented cryptographic routines can show significant timing differences depending on inputs [15]. This may not be apparent for typical neural networks, where each input follows the same path through the program code. However, input-dependent timing may be apparent in some algorithms (e.g. bypassing later layers of a neural network [11]).

Specific bit patterns may also affect timing. For example, in traditional (i.e. non-AI) computing, subnormal numbers can have a significant effect on execution time [3]. If the operational software is based on floating point numbers then bit patterns provide another aspect of measuring coverage across $\mathcal{I}$.

3.6 Logging

Learning from incidents is an important part of a good safety culture. Sufficient AI-related information needs to be logged to support this learning. From our perspective, logging raises two key questions.

Firstly, whether logging relies on features of the SoC or whether it uses other system features (e.g. recording inputs and outputs at the SoC boundary). If SoC features are used then logging may create a new interference path, or emphasise a previously-identified one.

Secondly, (assuming SoC features are used), whether logging-related demands are constant, or whether they vary depending on the prevailing situation. Varying logging demands are another factor that can affect WCET.

3.7 Safety Islands

A large-scale, complex SoC intended for embedded use within a safety-related context may include a "safety island". This is a specific set of isolated hardware that is dedicated to fault handling [7].

In the case of off-the-shelf hardware, full details of any safety island are unlikely to be available. Nevertheless, areas of potential interference between the safety island and the rest of the SoC should be identified. This is a special case of interference path analysis.

The safety island would be expected to detect and respond to failures elsewhere on the SoC. This functionality should be examined as part of test activities (e.g. by inducing failures in different parts of the SoC).

3.8 Hardware-Based Trojans

Hardware-based Trojans are an acknowledged potential vulnerability in software systems [1]. Consequently, they present a notable threat to the use of AI in safety-related systems.

Theoretically, full control of the entire supply chain is sufficient to protect against hardware-based Trojans. In reality, the size and dynamic nature of the supply chain mean this level of control is impossible. Supply chain monitoring is important, but a multi-layered argument is needed.

Thinking about the standard cyber security Confidentiality, Integrity and Availability (CIA) triad, availability should be detectable and manageable by traditional safety measures. From a system perspective, this is the same situation as a hardware failure of the SoC. Handling this situation will require system-level architectural features. If there is a means of checking the AI output then traditional safety measures should also be able to detect loss of integrity. If the output cannot easily be checked then, as before, system-level architectural features should protect against the hazard that "AI provides an undetectable incorrect result". Confidentiality is very difficult to protect. System-level architectural designs, which, from a confidentiality perspective, treat the SoC as an untrusted "black box" may be the most appropriate way of mitigating this risk.

3.9 System-Level Architecture

Activities to support the use of computational hardware in safety-related domains can be split between fault prevention and fault tolerance, with the latter subdividing into fault detection and fault recovery. (This represents a significantly simplified view of the concepts and taxonomy of dependable and secure computing [6].) Much of the previous discussion has focused on prevention. Tolerance is typically achieved through system-level architectural design.

Tolerance can be achieved by using some form of diversity. Historically, one option involved using multiple, independent software teams. Experience has shown there are difficulties with this approach: it is costly; and it is difficult to quantify its benefits, which might not be as much as first appears [14]. ML development approaches change the cost-profile of software development [5]. They replace some expensive human effort for potentially less-expensive compute power. This may make diversity cheaper to achieve. Furthermore, the large number of samples in the TTV data sets may make it easier to measure diversity.

Diversity could also be achieved by using different ML development tools [20] or using different AI-enabling hardware. Another form of diversity could be achieved by combining an AI channel with a monitor channel implemented using traditional software techniques [8]. This would reduce the assurance burden borne by the AI channel (and the AI-enabling hardware). However, defining a suitable monitor is a non-trivial task.

4 Summary

Some form of assurance argument will be needed to support the use of AI in safety-related applications. Considerations related to AI-enabling hardware, typically large-scale, complex, off-the-shelf SoCs, will be an important part of that argument. This argument needs to investigate the hardware in the context of the hosted software applications. In loose terms we require confidence that:

- It will work, in general;
- It will work, in unlikely situations;
- Errors (i.e. failures to work) will be detected;
- Information will be protected (i.e. security).

Information to support those assertions should be generated from a variety of activities. Examples (mapped to subsections of this paper) are indicated in Table 2: ★ marks activities that directly support an assertion; ☆ marks activities where support is indirect.

Table 2. Support provided by activities to assertions (★ indicates direct support, ☆ indicates indirect support).

Activity (Subsection)	Work, in general	Work, in unlikely	Detect errors	Protect information
History of previous use (3-Intro)	☆			
Document and justify config. (3.1)	★	★	☆	☆
Run-time checks on config. (3.1)	★	☆		☆
Microcode updates (3.1)	★	★		☆
Control debug features (3.1)	☆	☆		★
Theoretical interference (3.2)	☆	☆		★
Empirical interference: apps (3.2)	★	☆		
Empirical interference: enemy procs (3.2)	★	★		
Self-interference (3.2)	☆	★		
Effects of numerical precision (3.3)	☆	★		
Exhaustive coverage of TTV data sets (3.3)	★	★		
Sampled coverage of TTV data sets (3.3)	★	☆		
Simple host-target comparisons (3.3)	★	☆		
Test coverage of $\mathcal{I}$ (3.4)	★	☆	★	
Test coverage of $\mathcal{O}$ (3.4)	☆	★	★	
Test coverage of $\mathcal{F}$ (3.4)	★	★	☆	
Test coverage of $\mathcal{A}$ (3.4)		☆		★
Testing on each and every SoC (3.4)	★	☆		
Effect of specific inputs on WCET (3.5)	★	☆		
Effect of specific bit patterns on WCET (3.5)	★	★		
Potential effect of logging (3.6)	☆	☆		
Safety island independence (3.7)		☆	★	
Safety island functionality (3.7)		☆	★	
Partial control of supply chain (3.8)		☆		☆
Monitoring SoC manufacturer errata (3.8)		☆		★
Wrapping Intellectual Property (IP) cores (3.8)				★
Multiple AI channels (3.9)	☆	☆	★	
AI and non-AI channels (3.9)			★	

Collectively, these activities cover the assertions we wish to make. This provides some confidence that a compelling assurance argument can be made to support the use of AI-enabling hardware in a safety-related system. However, there is a danger that Table 2 can be interpreted too favourably. There is much work to be done before all of the activities are well understood and routinely implemented as part of general engineering practice. Whilst this paper provides a signpost towards a compelling argument, we still have some way to go before we reach that destination.

References

1. Adee, S.: The hunt for the kill switch. IEEE Spectr. **45**(5), 34–39 (2008)
2. Alexander, R., Hawkins, H.R., Rae, A.J.: Situation coverage-a coverage criterion for testing autonomous robots. University of York (2015)
3. Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., Shacham, H.: On subnormal floating point and abnormal timing. In: 2015 IEEE Symposium on Security and Privacy, pp. 623–639. IEEE (2015)
4. Ashmore, R., Calinescu, R., Paterson, C.: Assuring the machine learning lifecycle: desiderata, methods, and challenges. arXiv arXiv:1905.04223 (2019)
5. Ashmore, R., Madahar, B.: Rethinking diversity in the context of autonomous systems. In: Engineering Safe Autonomy, 27th Safety-Critical Systems Symposium, pp. 175–192 (2019)
6. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**(1), 11–33 (2004)
7. Bello, L.L., Mariani, R., Mubeen, S., Saponara, S.: Recent advances and trends in on-board embedded and networked automotive systems. IEEE Trans. Industr. Inf. **15**(2), 1038–1051 (2018)
8. Caseley, P.: Claims and architectures to rationate on automatic and autonomous functions. In: 11th International Conference on System Safety and Cyber-Security (SSCS 2016), pp. 1–6. IET (2016)
9. Certification Authorities Software Team (CAST): Multi-core processors. Tech. rep. CAST-32A. Federal Aviation Administration, November 2016
10. Chakraborty, A., Alam, M., Dey, V., Chattopadhyay, A., Mukhopadhyay, D.: Adversarial attacks and defences: a survey. arXiv arXiv:1810.00069 (2018)

11. Gopinath, D., Converse, H., Pasareanu, C.S., Taly, A.: Property inference for deep neural networks. arXiv arxiv:1904.13215v2 (2019)
12. Gysel, P., Pimentel, J., Motamedi, M., Ghiasi, S.: Ristretto: a framework for empirical study of resource-efficient inference in convolutional neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5784–5789 (2018)
13. Iorga, D., Sorensen, T., Donaldson, A.F.: Do your cores play nicely? a portable framework for multi-core interference tuning and analysis. arXiv arXiv:1809.05197 (2018)
14. Knight, J.C., Leveson, N.G.: An experimental evaluation of the assumption of independence in multiversion programming. IEEE Trans. Softw. Eng. **1**, 96–109 (1986)
15. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
16. Luckcuck, M., Farrell, M., Dennis, L., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: a survey. arXiv arXiv:1807.00048 (2018)
17. Nicopoulos, C., et al.: On the effects of process variation in network-on-chip architectures. IEEE Trans. Dependable Secure Comput. **7**(3), 240–254 (2008)
18. Pronzato, L., Müller, W.G.: Design of computer experiments: space filling and beyond. Stat. Comput. **22**(3), 681–701 (2012)
19. Ribeiro, M.T., Singh, S., Guestrin, C.: Why should I trust you?: explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1135–1144 (2016)
20. Srisakaokul, S., Wu, Z., Astorga, A., Alebiosu, O., Xie, T.: Multiple-implementation testing of supervised learning software. In: Proceedings of the AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems, pp. 384–391 (2018)
21. Szegedy, C., et al.: Intriguing properties of neural networks. In: Proceedings of the 2nd International Conference on Learning Representations, pp. 1–10 (2014)
22. Taram, M., Venkat, A., Tullsen, D.: Context-sensitive fencing: securing speculative execution via microcode customization. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 395–410 (2019)
23. Teh, J.S., Samsudin, A., Al-Mazrooie, M., Akhavan, A.: GPUs and chaos: a new true random number generator. Nonlinear Dyn. **82**(4), 1913–1922 (2015)
24. Varshney, K.R.: Engineering safety in machine learning. In: Proceedings of the 11th IEEE Information Theory and Applications Workshop, pp. 447–451 (2016)