Hardsec: practical non-Turing-machine security for threat elimination

Sandy Macadam
sandym77@gmx.com
hardsec.org

Abstract

Security based on threat detection is a never-ending game of cat and mouse. By contrast, strong cybersecurity eliminates threats without relying on threat detection. Strong cybersecurity requires however that the threat elimination platform itself can be trusted: if an attacker can compromise the threat elimination platform then there can be no confidence that it continues to carry out threat elimination. Software running on a CPU that is (approximately) a Turing machine is inherently hard to secure against compromise, but hardware security suffers from problems due to inflexibility and high up-front costs. We present a practical approach (which we describe as 'hardsec') which uses Field Programmable Gate Array (FPGA) devices to deliver effective security threat elimination using non-Turing-machine implementations.

Introduction

It is well known that maliciously constructed data inputs can be used by an attacker to exploit system vulnerabilities. Security approaches to defending against malicious data inputs fall into one of two camps. In one approach, steps are taken to detect malicious data inputs so that any such data is blocked before it is used as input. An attacker is of course motivated to avoid detection, and this approach therefore ultimately becomes a never-ending game of cat and mouse.

The second approach is to assume that all potential input data is malicious, and thus take steps to sanitise the data before it is used. This approach presents an inherent paradox, because the data must necessarily be used as input to the threat elimination system that carries out the sanitisation, and so could be used to exploit a vulnerability in that threat elimination system. Once the threat elimination system itself is exploited, the attacker can then output malicious data designed to exploit the original system.

One way of addressing this paradox is to run multiple independent threat elimination systems in series: an attacker then needs to prepare and deliver multiple different malicious datasets designed to exploit each of the threat elimination systems in turn. This increases the level of effort and cost that the attacker must expend in order to achieve their ultimate goal, which is to exploit the original system. However, it also increases the level of cost to the defender, because they must procure multiple independent threat elimination systems. It is not clear which of these costs increases the most rapidly: the cost to the attacker or the cost to the defender. In addition to the defender's direct costs in procuring multiple independent threat elimination systems, there is a further cost in some circumstances in that there will be an impact on end-to-end latency which can cause undesirable impact for the end user.

The other way of addressing the paradox is to engineer threat elimination systems using techniques which make them harder to exploit than the systems they are designed to protect. This has historically been the goal of high assurance software engineering techniques. After briefly reviewing the challenges with these techniques, we present an alternative approach based on the use of non-Turing-machine implementations.

High assurance software engineering

The critical challenge with engineering high assurance software is that the CPUs on which software runs are (almost, given the constraints of finite memory) a form of universal Turing machine. The infinite flexibility of a CPU is at the heart of the information technology revolution, but is also its Achilles' heel from a security perspective. Because a CPU is capable of (almost) any possible function, the requirement for high assurance software engineering is to demonstrate that the loaded software will behave "as expected" for any possible input. If this goal is not achieved, it may be possible for a suitably crafted input to cause the CPU to behave in a completely unexpected manner – ie, it may be able to exploit a vulnerability in the system.

Engineering high assurance software is known to be problematic for two reasons.

Firstly, when engineering high assurance software, productivity is low. A range of techniques have been developed for verifying the correctness of both a design (ensuring that "as expected" is defined for every possible input) and an implementation, but even after decades of innovation, the process is very substantially slower and more costly than mainstream software engineering methods.

Secondly, even correctly generated high assurance software will fail to achieve its aim of preventing system exploitation if there is a vulnerability in the underlying CPU platform on which it is running. Commonly used CPU platforms such as those based on the Intel x86 architecture have historically been found to have a number of vulnerabilities (Meltdown and Spectre being the most widely publicised but see also [1], for example). Progress has been made in creating formally verified CPU designs that can be proved not to contain vulnerabilities, but to date commercial silicon implementations do not exist.

Non-Turing-machine implementations

The reason that these costly high assurance engineering techniques are required to make a software system hard to exploit is that, if the software is not correct, the potential results are dramatic. Because a CPU is a form of universal Turing machine, an error in the software that presents a vulnerability could result in the attacker being able to run their own software instructions (malware) on the system. This malware can be dramatically different from the original software and might run instead of, or in addition to, the original software.

This analysis suggests an alternative approach to making systems robust against exploitation. Rather than seek to engineer an error-free system, we could seek to engineer a system where the potential results of an error are less dramatic. This can be achieved if, instead of using a CPU, a system is engineered using non-Turing-machine implementations which are capable of only a much narrower range of functions. In this case, even if there is an error in the implementation, any malicious input will have a much narrower set of outcomes which it can generate.

Automata theory defines a hierarchy of computational power, with a Turing machine being the most powerful. At the other end of the scale, pure combinatorial logic is the least powerful; slightly more powerful are finite state machines and deterministic finite automata. Various classes of more powerful automata complete the gap.

Systems that depend purely on the lowest-power computational approaches provide lower power to the system developer, but equally provide lower power to an attacker. With a finite state machine for example, it is certainly possible for the state transition table to be incorrectly defined, but the

range of possible outcomes that result is very tightly bounded, provided that the table itself cannot be overwritten by the inputs (a critical requirement to which we return below).

This approach has been the field of hardware security. Modern processors based on architectures such as x86 and Arm incorporate a range of hardware security features such as ring protection and virtual machine separation that are based on non-Turing-machine logic. However, these practical implementations of hardware security suffer from two related problems.

Firstly, because the security feature is implemented in hardware, there is no way that it can be modified post-manufacture in order to resolve a problem. For example, several Arm architecture platforms implementing the Arm Trustzone security feature have been found to have hardware implementations that allow the Trustzone security to be compromised (e.g. [2]). There is no way to "patch" these vulnerabilities at the hardware level.

Secondly, because these platforms' hardware security features cannot be changed post-manufacture, they are often designed to be highly configurable by software. This then exposes them to subversion by compromised software (this has been seen in practice with numerous security subsystems in Intel platforms such as [3] and [4]).

In addition to these problems, the high up-front cost of manufacturing integrated circuits means that traditional hardware security has only been applicable for extremely widely-encountered requirements (such as ring protection). Most security requirements cannot justify the up-front cost of developing and manufacturing dedicated silicon.

A more practical approach to non-Turing-machine security

Non-Turing-machine security can be implemented while avoiding these pitfalls by using Field Programmable Gate Arrays (FPGAs). FPGAs are a widely available form of integrated circuit which allow digital logic circuits to be programmed and then reprogrammed repeatedly in the field, long after the device was manufactured. FPGAs avoid the pitfalls with existing hardware security implementations because they are reprogrammable in the field. If an implementation is found to be incorrect, or if new features are required, the FPGA can be reprogrammed.

However, because the FPGA can be reprogrammed, it is necessary to consider how to prevent an attacker from doing so. The answer is to depend on the physical arrangement of the platform within which the FPGA is deployed. Reprogramming of FPGAs takes place through specific physical connections to the FPGA die, usually exposed as pins on an FPGA package. If an attacker can be physically prevented from transmitting data to these pins, they are unable to reprogram the FPGA.

A practical FPGA 'hardsec' implementation therefore has a management pathway which is physically separate from input and output pathways. Only the management pathway is able to reprogram the FPGA, and an attacker having access only to input or output pathways cannot do so. Of course, in practice system operators will wish to minimise the frequency with which they need to physically visit any system. This can be achieved by accessing the management pathway via an out-of-band management network, the use of which is already standard practice for data centre equipment.

It is worth noting that although not widespread, the use of FPGAs in this manner is well established. High-security ('government-grade') cryptographic devices have been based on the use of FPGAs for many years (e.g. [5]). What is newer is the use of FPGAs for threat elimination.

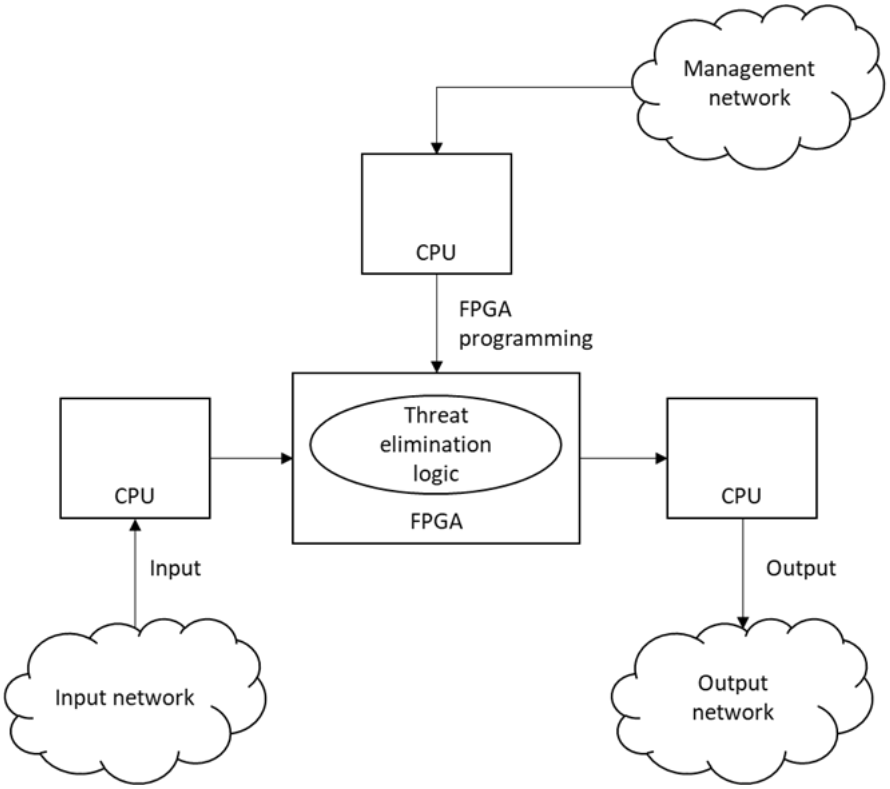

*Figure 1 – threat elimination with FPGA hardsec**

** Note that although three separate networks are shown, these networks could be physically collapsed using suitable encryption implementations, either on the CPUs shown or using additional hardware (to mitigate the risk of compromise to software cryptography implementations)*

Developing threat elimination logic for FPGA hardsec

It is generally easier to develop functionality using software languages than using hardware description languages such as VHDL or Verilog. At first sight, this would appear to be a significant drawback for FPGA hardsec approaches.

In some cases, faced with such challenges, electronic engineers may consider using a hardware description language to implement a "soft" CPU within an FPGA, and then developing software for that soft CPU. For hardsec, this is an invalid approach, because the soft CPU reintroduces the Turing machine architecture that hardsec is designed to replace.

Instead, threat elimination functionality with hardsec can usually best be implemented using a "transform and verify" approach as originally described by the United Kingdom's National Cyber Security Centre (NCSC – a part of the UK's GCHQ intelligence agency), in their "Pattern for Safely Importing Data" [6]. Software running on a CPU is used to transform data, and this transformed data is then sent as input to an FPGA where it is verified. For example, if a date field in the format ddd-mmm-yyyy is to be sanitised, software could be used to transform the date field to an integer number (for example, number of days since 1st January 1970) and this integer number will then be verified using logic in the FPGA. Using a hardware description language (HDL), it is much easier to develop logic to verify an integer than to verify the original date format. Once the verified (and hence safe) data has been output from the FPGA, it can be transformed back into the original format by software.

As an alternative to developing the verification functionality using an HDL, the HDL could be used to implement a low-power automaton (typically a Deterministic Finite Automaton) with the verification logic being defined as a state transition table for the DFA. To maintain the hardsec benefits, the DFA state transition table must only be programmable via the management pathway, along with the programming of the logic into the FPGA. With reference to [6] it is notable that in addition to the "verify" part of the pattern, FPGAs are also ideally suited for implementing the "flow control" part of the pattern.

A further benefit: functional decomposition with FPGAs

It is common for any practical logic implementation (for example, software) to incorporate a functional decomposition such that the overall functionality is achieved by combining multiple individual functions. However, the Turing-machine nature of a CPU platform is such that compromise of any one such software function could potentially lead to compromise or bypass of the software logic that enforces the functional decomposition, or to compromise of other functions in the decomposition.

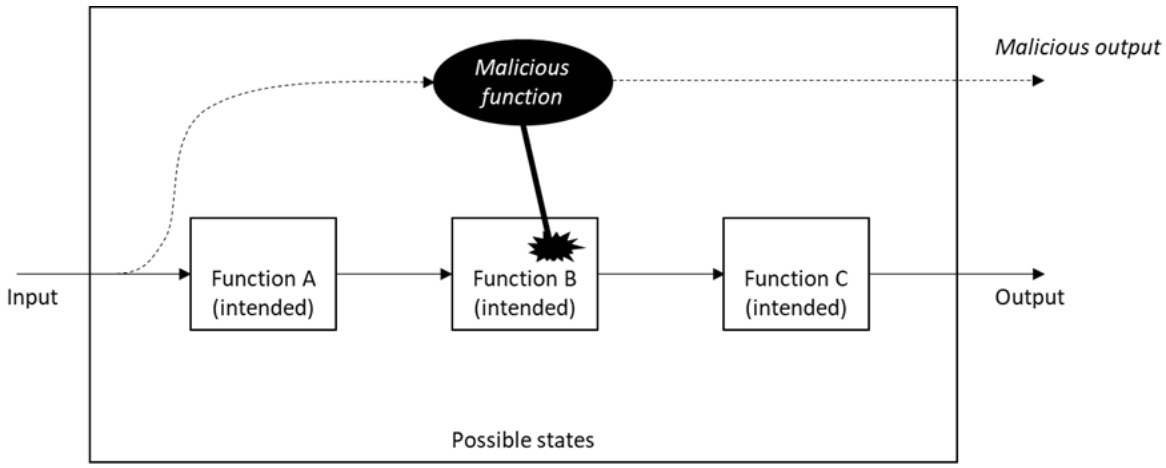

Figure 2 – possible catastrophic effect of exploiting a vulnerability in a software implementation

With an FPGA implementation, the decomposition into functional blocks is carried out through simple wiring that connects distinct logic blocks. If an error in one of the functional logic blocks were to be in some way exploitable (even, for example, if that block were to implement a soft CPU which at some point started running malware) it would not be possible for that logic block to modify the wiring that enforces the functional decomposition, or to modify the functioning of any other logic block in the decomposition. This confidence in the decomposition provides immediate benefits at the top level: we can be confident that even if function A is compromised, function B will still be applied. However, functional decomposition can be applied iteratively, breaking all functionality down to the simplest levels in order to increase confidence throughout the design even in the absence of a formal proof.

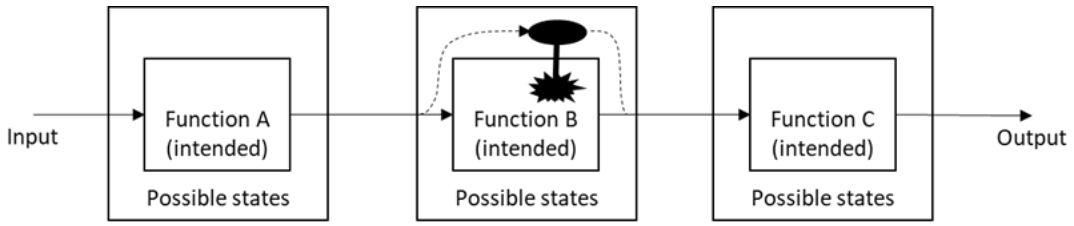

Figure 3 – limited effect of exploiting a vulnerability in an FPGA implementation

Hardsec architectures

There are good reasons why the world is built on software and (near) Turing-machine CPUs. It would not be a practical suggestion to try and replicate all that can be achieved with software using hardware description languages. Yet hardware security already underpins all of this, through hardware processor features such as ring protection and virtual machine separation.

There are however clear problems with both software and hardware security. We have presented an alternative 'hardsec' architecture which overcomes many existing problems. The three key characteristics of this architecture are: the use of non-Turing-machine implementations; field upgradability; and a management (programming) pathway which is separated from any input pathway. The architecture can be practically implemented using FPGA technology.

This hardsec architecture can further be combined with the 'transform and verify' approach introduced by the UK's NCSC [6] to provide a robust and practical platform for threat elimination. Real world implementations of hardsec exist in the market today for some initial use cases.

References

[1] Intel®, "Intel Q3'17 ME 6.x/7.x/8.x/9.x/10.x/11.x, SPS 4.0, and TXE 3.0 Security Review Cumulative Update",
https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00086.html

[2] Adrian Tang, Simha Sethumadhavan, Salvatore Stolfo, "CLKSCREW: Exposing the perils of Security-Oblivious Energy Management",
https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-tang.pdf

[3] Rafal Wojtczuk, Joanna Rutkowska, Alexander Tereshkin, "Another way to circumvent Intel® Trusted Execution Technology",
https://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf

[4] Intel®, "Intel Active Management Technology, Intel Small Business Technology, and Intel Standard Manageability Escalation of Privilege",
https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00075.html

[5] Airbus Defence and Space Inc, "Ectocryp® Blue",
http://ectocrypusa.com/blue/ECTOCRYP-Blue-overview.asp

[6] UK NCSC, "Pattern for safely importing data",
https://ncsc.gov.uk/guidance/pattern-safely-importing-data