Hardware Trojan Resistant Computation using Heterogeneous COTS Processors

Mark Beaumont **Bradley Hopkins** **Tristan Newby**

Defence Science and Technology Organisation
Adelaide, Australia
Email: {mark.beaumont, bradley.hopkins, tristan.newby}@dsto.defence.gov.au

Abstract

Hardware Trojans pose a credible and increasing threat to computer security, with the potential to compromise the very electronics that ostensibly provide the security primitives underpinning various computer architectures.

The discovery of stealthy Hardware Trojans within Integrated Circuits by current state-of-the-art pre- and post-manufacturing test and verification techniques cannot be guaranteed. Therefore electronic systems, especially those controlling safety or security critical systems should be designed to operate with integrity in the presence of any Hardware Trojans, and regardless of any Trojan activity.

We present an architecture that fragments and replicates computation over a pool of Commercial-Off-The-Shelf processors with widely heterogeneous architectures. Processors are loosely synchronised through their use of a voted, architecture-independent message box mechanism to access a common memory space. A minimal Trusted Computing Base abstracts the processors as a single computational entity that can tolerate the effects of arbitrary Hardware Trojans within individual processors. The architecture provides integrity, data confidentiality, and availability for executing applications.

1 Introduction

Hardware Trojans are malicious modifications to Integrated Circuits (ICs) that can compromise the security of a hardware platform or any software running on it. They are persistent in nature and can operate continuously or be triggered into one or more actions, including modification of functionality, modification of specification, leaking of sensitive information, or Denial of Service (DoS) (Rajendran et al. 2010). The severity of Hardware Trojan action can range from minor through to catastrophic, such as the disabling of a major financial system causing economic loss or affecting a critical electro-mechanical system (Tsang 2009), leading to potential loss of human life.

Many different types of Hardware Trojans have been demonstrated, (Lin, Burleson & Paar 2009)(Baumgarten et al. 2011)(Jin et al. 2009), including malicious modifications to CPUs that have enabled privilege elevation and password stealing attacks (King et al. 2008). Whilst not a malicious Trojan, the Intel Pentium `f00f` bug (Collins 1998) demonstrated how a small design flaw in an IC could render a system vulnerable to a DoS attack.

The Hardware Trojan threat is increasing, especially amongst Commercial-Off-The-Shelf (COTS) components, where much of the IC development chain has been outsourced, relinquishing control over many potential Hardware Trojan insertion vectors. The past six years have seen increased research into methods for detecting Hardware Trojans. The primary methods involve self-checking systems, side-channel analysis and destructive reverse-engineering (Chakraborty et al. 2009). Even with the most recent advances in detection techniques, there are no guarantees that an IC is free of Hardware Trojans (Abramovici & Bradley 2009).

Economic and political rationale are driving increased globalisation, pushing *untrusted* COTS components into many electronic devices, including safety critical systems and sensitive military equipment (Young 2011). The cost of developing a Trojan-free IC is immense, requiring trust in many areas including design tools and teams, fabrication facilities, supply chains and anti-tamper technology. This approach is currently both technologically and economically infeasible, especially in an Australian Military context. To track technological advances, especially in relation to the latest processor architectures, accreditation of all components is not practicable, thus the use of COTS elements cannot be avoided. Instead, we advocate coupling the latest COTS technology with some small, accreditable trusted logic to form a Trojan-hardened system.

In previous work, the SAFER PATH architecture (Beaumont et al. 2012), a Hardware Trojan-resistant general computing platform, was proposed as a trusted drop-in replacement for a potentially compromised processor. The architecture combines many similar, cycle-accurate processors with a small Trusted Computing Base (TCB) to achieve replicated and fragmented execution. The architecture provides integrity and availability through majority voting of execution and protects data confidentiality by limiting any individual processor's access to program code and data. It relies on obtaining variations of the same processor for protection against identical Trojans.

In this paper, we introduce a modified version of the SAFER PATH architecture that abstracts a single computational entity from the collective behaviour of a pool of COTS Processing Elements (PEs) with *widely heterogeneous* architectures. Computation across multiple PEs is loosely synchronised via an architecture-independent message box (mbox) mechanism, allowing voted execution of an application. This execution is also fragmented in time across many different sets of PEs, limiting access to sensitive infor-

Copyright ©2013, Commonwealth of Australia. This paper appeared at the Thirty-Sixth Australasian Computer Science Conference (ACSC 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 135, Bruce H. Thomas, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

mation for any individual PE. A software interpreter is demonstrated that provides a software abstraction, allowing a single, architecture-independent application to be collectively executed and fragmented across a pool of architecturally different processors.

The replication and fragmentation logic are part of a minimal TCB, providing the root of trust for the architecture. As such, effort must be put into ensuring that this logic is free of Hardware Trojans. Development and accreditation of the TCB logic is far more economically feasible than pursuing a complete trusted processor.

Trustworthiness is targeted at the expense of cost, performance, power usage and size. This is a design decision, but one that we believe needs to be made, especially for critical systems.

Our focus is on Hardware Trojans present within PEs, e.g. a CPU with local memory, and we aim to provide a broad spectrum defence against the effects of any Trojans present within these circuits. While we assume that any given PE may be infected, the likelihood of having identically functioning, or collaborative Hardware Trojans across many processors is considered very low, decreasing as the number of different processors is increased.

External to our abstracted computational entity, we provide no protection against Hardware Trojans in other ICs such as system-wide memory, or Input/Output (IO) circuitry. The architecture ensures that any given software is executed correctly as determined by the collective behaviour of multiple PEs. This architecture can be used as a trusted replacement processing element, with other defences able to be incorporated to protect the system as a whole, e.g., Bloom et al. (2009) protect against Trojans residing in memory using a double guard on the memory bus.

The paper is organised as follows: Section 2 discusses related work, Section 3 details our proposed solution and Section 4 describes our experimental implementation and results. Section 5 discusses some potential extensions while Section 6 summarises our work.

2 Related Work

There are existing commercial and industrial systems that provide availability, and protect functional integrity and data confidentiality. They often incorporate one or more of the following techniques: heterogeneous processors, redundant processing, software dissimilarity, voting, and data fragmentation. Hardware Trojan research has also incorporated some of these mechanisms.

Recently, the SAFER PATH architecture was developed incorporating fragmented execution and replication as a defence mechanism. SAFER PATH relies on obtaining variability between operationally equivalent processors to combat Hardware Trojans. Ensuring there is enough variability between processors to prevent the same Hardware Trojan appearing is difficult, requiring sufficient orthogonality between the design teams, design software, and fabrication facilities. It is also difficult to obtain this variability off-the-shelf, meaning that processors would need to be customised. Utilising a new type of processor would require significant effort. The same Hardware Trojan might also be more easily inserted into variants post-manufacture, given that all processors must adhere to the same operational interface.

In contrast, the architecture presented here uses truly heterogeneous, unmodified COTS processors, allowing new types of processors to be easily added. Every processor in the architecture can be different, increasing the barrier for any collaborative Hardware Trojan insertion.

Yeh (1996) describes the use of triple modular redundancy using heterogeneous PEs, majority voting and N-version dissimilar software to achieve high levels of reliability in the Boeing 777 primary flight computer. A low-level communications bus is used for synchronisation between varying processor channels. The system is only used to process simple inputs and outputs, and only outputs are voted on. Saxena and McClusty (1998) use redundant simultaneous multithreading to achieve fault detection and recovery at a software level. In more recent work, Reis et al. 2005 employ compiler-based transforms that duplicate instructions and insert checkpoints for fault detection. These systems provide protection against transient faults, as opposed to Hardware Trojans which may not manifest as a fault, but rather a subtle change to a processor's behaviour, or the leaking of sensitive information.

McIntyre et al. (2010) propose a software fault-tolerant technique, the Trojan Aware Distributed Scheduling (TADS) system. TADS operates on a multi-core compute platform potentially containing one or more Hardware Trojans. A scheduler is used to execute functionally equivalent subtask variants on different cores within the processor. Results are evaluated for equivalence and any disparity is used as an indicator of Hardware Trojan presence. This process is repeated and the scheduler is able to progressively establish trust in the circuitry of each core. Software variants provide course-grained protection and require program diversity through recompilation. This architecture protects against simple Hardware Trojans, but is vulnerable to more sophisticated Trojans (e.g., King et al. 2008) that may be replicated across processing cores.

Other research has proposed Data Guards (Bloom et al. 2009) (Waksman & Sethumadhavan 2011) and reconfigurable logic (Baumgarten et al. 2010) to counter the presence of Hardware Trojans within ICs. These solutions are focused on protecting against specific Hardware Trojan triggers or actions. We make no such assumptions about the type of triggers that may exist or the actions that may result, and presume Hardware Trojans may be active within all our processors.

3 Architecture

Modern computing systems typically entrust one or more COTS Processing Elements (PEs) to reliably execute programs. Our assumption is that any of these individual PEs may be infected by one or more Hardware Trojans, consequently compromising security by modifying the behaviour of the program or leaking data.

Our architecture uses a pool of many architecturally different PEs together with a small Trusted Computing Base (TCB), to collectively execute a given application. The architecture enables the external behaviours of simultaneously executing PEs to be supervised, making no assumptions about the internal operation of individual PEs. All PEs run independently of each other, executing their own code from a locally attached memory. Low-level computation is not replicated, instead, some of the external behaviours of the PEs are replicated and arbitrated by the TCB to coordinate a collective behaviour across the pool of PEs.

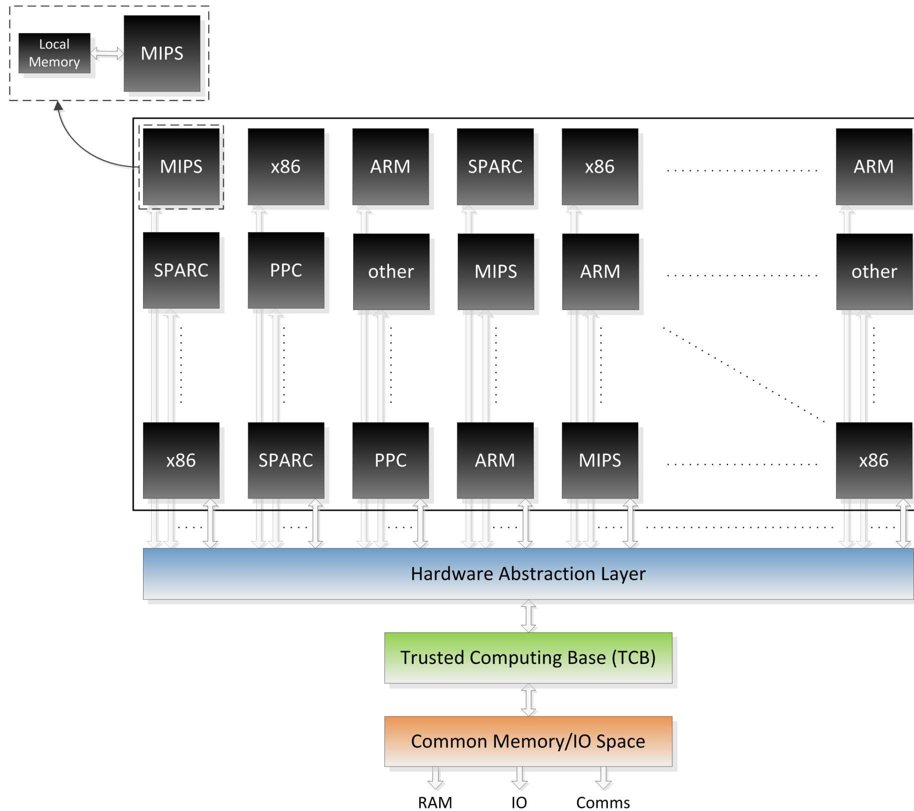

Figure 1: A set of PEs collectively execute a given application.

To support this behavioural replication across different architectures, a Hardware Abstraction Layer (HAL) provides independent access to a common memory and IO space. The HAL unifies accesses using a message box (*mbox*) associated with each PE. Mboxes provide a register-style interface to access the common resources. The architecture is shown in Figure 1.

Software applications are independently compiled for each PE architecture, and all accesses to the common memory and IO space are made through architecture-specific mbox routines. The compiled applications execute concurrently across a set of PEs, but execute different machine code and have different mbox access timing. To obtain collective behaviour from a subset requires the mbox accesses to be strictly ordered. At any instance in time, execution is loosely synchronised across a set of PEs, enforced by the TCB which arbitrates access to the common memory space using a simple voting mechanism.

The TCB also facilitates time-domain fragmentation of program behaviour across multiple independent sets of PEs from the pool, protecting against side-channel data leakage attacks (Lin, Burleson & Paar 2009) (Lin, Kasper, Paar & Burleson 2009). Application synchronisation between sets is maintained by storing selected elements of program state in the common memory.

3.1 Processing Elements

Our architecture supports the use of almost any type of COTS processing element, e.g., ARM, MIPS, x86, SPARC, to form a large resource pool. This architectural diversity minimises the probability of colluding or replicated Hardware Trojans existing within different PEs.

Independently infected PEs alone cannot compromise the integrity of the computation; an adversary would need to influence multiple designs, fabrication facilities, or supply chains to bypass this diversity. New types of PEs can always be incorporated into the architecture to maintain this diversity and track technological developments.

3.2 Message Boxes

Heterogeneous PEs have different bus interfaces, timing characteristics and byte orderings, making it difficult to combine their behaviours, especially at a low level. The mboxes ensure each architecture can access the common memory and IO space, facilitating synchronisation and voting. The mboxes form a trust boundary between a PE and the TCB. A simple interface enables easy system integration, promotes simple TCB design and allows the architecture to scale to a large number of PEs.

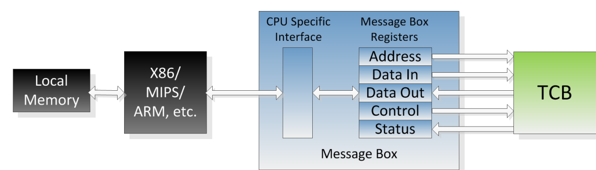

Figure 2: mbox register interface.

Access to common memory and IO is abstracted through a transaction style approach using registers for address, data, control and status as shown in Figure 2. When a PE wishes to read or write to the common memory, it writes the address (together with the data for a write) and then signals through the control register for the operation to be performed. The TCB decodes the address and forwards the request to the appropriate memory or IO resource. Once the

operation is complete (and data returned for a read) a status register bit is set.

Individual PEs could potentially communicate with an mbox using any available communication interface, for example a memory bus, GPIO port, or USB or PCIe interface. However, the implementation must consider potential Hardware Trojan interference. Strict separation between mbox communication channels must be maintained, and, where mboxes are not part of the TCB, they also require diversity in their design and manufacture to counter replicated or colluding Hardware Trojans.

3.3 Loosely Synchronised Execution

The TCB interfaces with the mboxes and combines the multiple access requests into a single collective request to the common memory and IO space. Read accesses result in the same data being returned to all PEs. Figure 3 shows a set of PEs writing data to a common memory address; specifically shown is the asynchronous nature of the requests from the different PEs.

The same, single application is independently compiled for and subsequently run on each PE. Although compilation is generally from the same source code, there may be significant differences in the respective machine code representations. Loose synchronisation between the executing programs is maintained by ensuring each PE attempts the same mbox accesses in the same order. This strict ordering is enforced during the software development for the architecture, with consideration given to different architectures, programming languages, and compilers. The TCB generally blocks on these access requests until all PEs have updated their respective mboxes. The TCB then votes on the requests and performs the actual memory access to the common memory space. This synchronises execution of the application on all mbox operations and ensures voting occurs on the same accesses.

Voted output guards against Hardware Trojans attempting to modify program behaviour. It also prevents leakage of confidential data through logical channels, either common memory or IO. While we implement a majority voting mechanism in our concept demonstrators (Section 4), different access aggregation policies may be used. Individual PE accesses may be weighted or even ignored by the TCB when generating the voted output. Such voting schemes can help protect against potential DoS attacks. The voting algorithm can be adaptive and designed with potential risk profiles for different PE architectures taken into account. Changes to the TCB need to balance performance and complexity.

3.4 Fragmentation

Even when the direct outputs of a collective program are protected, Hardware Trojans may still be able to leak data through side-channels. To combat this, the execution of the collective program, i.e. its behaviour, is fragmented in time across many different sets of PEs. A given set runs a *fragment* of the collective program before execution of that program is switched to another set of PEs. Fragmentation of program execution is achieved by transferring the currently executing program context from one set of PEs to a different set of PEs.

In contrast to the SAFER PATH architecture, an individual PE's program code itself is not fragmented. Instead, all PEs in the architecture run continuously, however only a subset have access to the common memory and IO space at any given time. The TCB assembles these subsets of PEs and enables or disables their access to the common memory.

Application instances, on any particular PE, are informed when they become connected to the common memory space. Application support for fragmentation involves the unique instances running on

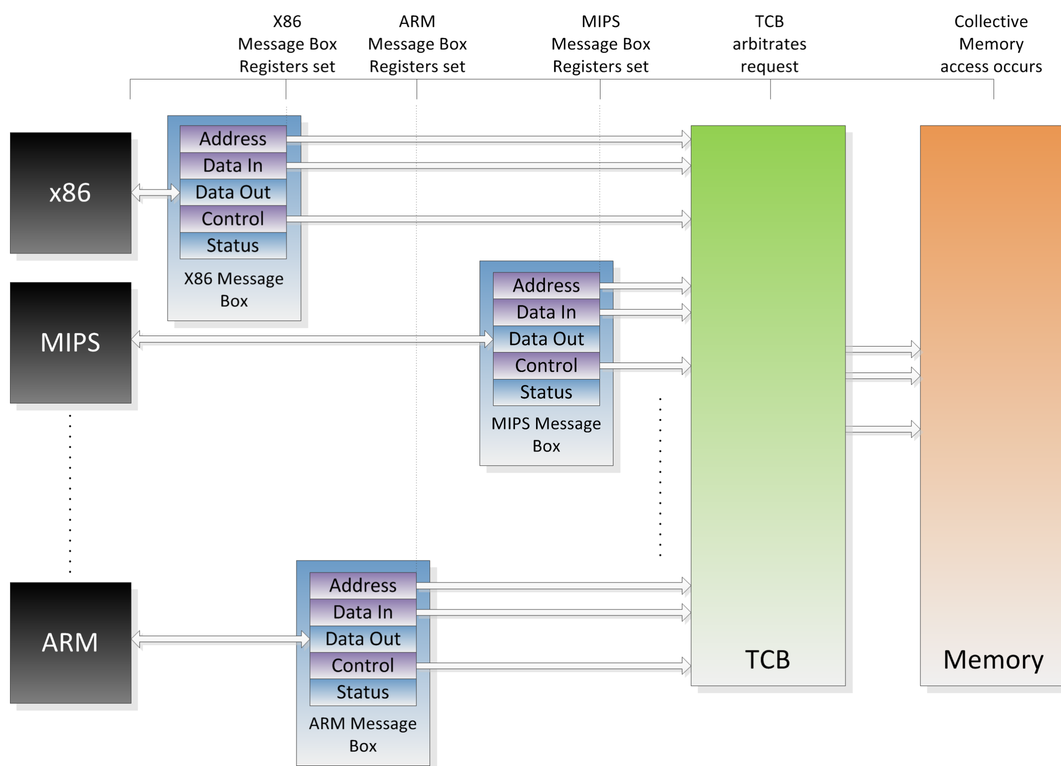

Figure 3: The TCB arbitrates a write access to common memory.

the currently active set of PEs saving collective program context to the common memory. The TCB can then switch its mbox interfacing to a new set of PEs. The application instances running on the new set of PEs load the saved context from the common memory and continue the collective execution. Saving and loading of this program context is implicit if all necessary program variables are permanently stored and accessed through the TCB protected common memory. A context switch can either be initiated from the software application, or mandated at periodic intervals by the TCB.

This form of context switching restricts an individual PE's access to sensitive data, limiting what information a potentially infected PE may leak. It also restricts any Hardware Trojans from understanding the broader scope of executing applications, making it difficult to interpret what data is currently being processed by a PE, and thereby increasing the complexity requirements of such Trojans. For example, execution may be fragmented to restrict individual PE access to an encryption key or sensitive report.

3.5 Trusted Computing Base

The TCB contains minimal logic to enforce the collective behaviour. A benefit of our architecture is that the TCB is a small, generic design that can be used with many different PEs, providing a more tractable and flexible solution than developing a custom trusted processor. While the TCB provides integrity, availability and confidentiality, the outputs are not necessarily correct; rather they reflect collective behaviour. As the number of PEs is increased, the outputs become probabilistically correct and more resistant to Hardware Trojans.

The design consists primarily of voting and switching logic plus additional ancillary circuits for the purpose of enforcing time-windows on mbox accesses. The simplicity and small size of the TCB relative to an individual PE assists both accreditation and subsequent design and fabrication free of Hardware Trojans. The TCB may include the mboxes or just an interface to the mboxes. This decision is dependent on obtaining variant mboxes that do not need to be trusted.

The TCB must also prevent misuse of the architecture. Rogue PEs may delay or insert additional mbox accesses in an attempt to degrade service. The heterogeneous processing nature of the architecture requires the TCB to aggregate accesses that are asynchronous. The TCB can use a time-window to ensure timely access synchronisation. If PEs violate this timing they can be blacklisted and removed from the set, or in the worst case, the TCB can reset all processors. A larger pool of PEs reduces the influence of this issue.

In our proposed architecture, the TCB arbitrates access to common memory and IO devices. Other peripherals could also be supported, such as system timers and interrupts, to enhance software application support, and enable more complete systems to be protected by the architecture. The trade-off for this convenience is the size and complexity of the TCB.

4 Experimentation and Results

The architecture was prototyped within a Xilinx Virtex 6 FPGA. A pool of embedded soft-core processors, an mbox for each processor, the TCB logic, and the common memory were all implemented within the FPGA. Three different processor architectures were used: *leon3* (Aeroflex Gaisler AB 2010), a 32-bit SPARCv8 processor; *mblite* (Kranenburg & van Leuken 2010), a 32-bit MIPS based processor; and *zpu* (Zylin Consulting 2008), a tiny, 32-bit stack based processor.

Each type of processor was configured with enough local memory (*leon3*: 16kB, *mblite*: 16kB/16kB, *zpu*: 32kB) for the example programs to run natively.

The mboxes associated with each processor were connected using a GPIO port native to each architecture. Each mbox consisted of a 32-bit address port, a 32-bit data-in port, a 32-bit data-out port and a control/status port. The mapping of memory and IO peripherals in this address space is application-specific.

Developing an application for the architecture requires identifying important information or computational actions to protect. Important computations need to be replicated and voted upon. Likewise, sensitive information must be fragmented across different sets of processors. For a custom application this normally entails voting on all accesses to the system inputs and outputs and supporting the fragmentation of the application across multiple subsets of processors. To achieve this, each natively executing processor in the currently executing set must perform the same common memory accesses in the same order.

Two example programs were developed. The first is a software interpreter where the interpreted program is stored and accessed through the TCB protected common memory. The output of the interpreted program occurs through a serial port, which is also mapped into the TCB protected common memory space. The second example program is a VNC client, where the network IO, keyboard and mouse inputs, and framebuffer outputs are mapped into the common IO space and protected by the TCB.

4.1 Software Interpreter

A consequence of using many different processor architectures is that programs need to be compiled for each architecture. To alleviate this requirement, we employed a software interpreter. Though the interpreter itself runs natively on each processor and, hence, needs to be compiled for each different architecture, once this has been done, many different programs can be run on top of this interpreter without needing to be rewritten or recompiled for the underlying architectures. A secondary benefit of the interpreter is that it also allows computation to be voted upon and fragmented, in addition to just the system IO. The fragmentation is trivially achieved because the interpreted program and interpreted state is entirely stored and accessed through the TCB protected common memory.

We utilised a pool of 12 processors divided into four sets, with each set containing one *mblite*, one *leon3*, and one *zpu* processor, each running at 25MHz. Input and output to the architecture is provided by an asynchronous serial port that is mapped into the common IO space. Synchronisation and fragmentation are supported by a 16kB TCB protected common memory.

The *ubasic* (Dunkels 2007) BASIC interpreter was ported to the each native processor architecture. The native code for each interpreter is stored in and executed from the local memory attached to each processor, i.e., 12 separate instances running in our experimental set up. Collectively, the processors interpret a single, architecture-independent BASIC program. The BASIC program to be interpreted (e.g., Listing 2), is stored in the common memory space. The *ubasic* implementation was modified so that calls to

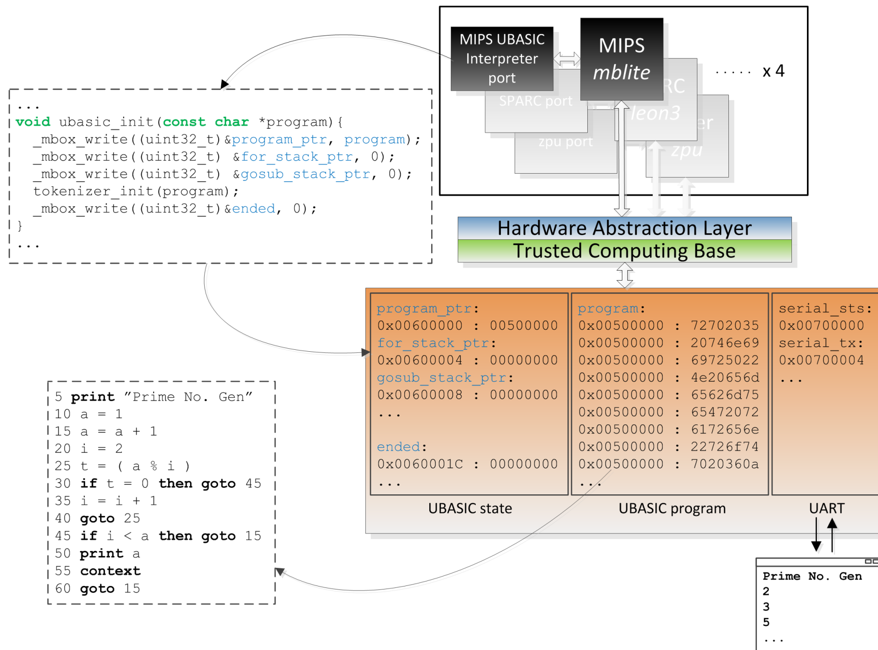

Figure 4: Operation of the software interpreter.

memory accesses associated with the BASIC program were replaced with accesses through the mboxes.

Simple asynchronous serial output was also provided through the common memory space. The BASIC program was able to write out this serial port using the *print* keyword. An instructive example of accessing the serial port via an mbox is given in Listing 1. When the *print* keyword is interpreted, the `outbytem` function is called for each character to be printed. The `_mbox_read` and `_mbox_write` calls ensure that reads from the serial port status register, and writes to the data (output) register are synchronised and voted on across the currently active set of heterogeneous processors.

```
void outbytem (char c) {
  do {
  } while ((_mbox_read(STATUS_REG) &
            BIT_SET(SERIAL_XMIT)) == 0);
  _mbox_write(SERIAL_REG, c);
}
```

Listing 1: Mbox access to a collectively controlled serial port.

In the demonstrator, all processors concurrently execute the *ubasic* interpreter, however, only a selected set of processors are connected to the common memory and thus interpret the instructions of the BASIC program at any one time. The *ubasic* software running on processors that are not connected remains in a waiting state until their connection to the common memory is restored. A read of a status bit through the processor's mbox indicates whether the common memory is connected. Once connected, the interpreters exit their waiting state and continue interpreting the BASIC program.

Enabling fragmentation of the BASIC program required further modification of *ubasic* to allow the execution context of the interpreter to be passed from the current subset to the next subset of processors. To support saving and loading of this execution context, some of the interpreter's state, including the program counter (a pointer into the BASIC program), stack pointers and global variables, is stored in and accessed through the common memory. A new keyword, *context*, was added to the *ubasic* instruction set that allows software to initiate fragmentation to a different set of processors. This is achieved via a write to an mbox control register bit. As with all mbox accesses, this write is voted on before being performed by the TCB, ensuring the context switch is a collective request.

The operation of the software interpreter is shown in Figure 4. A subset of processors connected to the common memory space accesses and interprets the BASIC program. The keyword *context* initiates a context switch to a new subset of processors and finally the new processors continue interpretation of the BASIC program. In this example, the program counter stored in the common memory is utilised to pick-up execution where the previous processors finished.

4.1.1 Performance

An example program (Listing 2) calculates prime numbers in a simple manner, performing an execution switch (line number 70) after printing the value of each successive prime.

```
5  print "Prime No. Gen"
10 a = 1
15 print "2"
20 a = a + 2
25 i = 2
30 t = (a % i)
35 if t = 0 then goto 60
40 b = a / i
45 if i > b then goto 65
50 i = i + 1
55 goto 30
60 if i < a then goto 20
65 print a
70 context
75 goto 20
```

Listing 2: BASIC prime number generator.

The performance of the *ubasic* interpreter on our architecture was analysed, using the BASIC program in Listing 2. The blocking nature of the mbox calls means a side-effect of loose synchronisation is to reduce the performance of the architecture to that of the slowest processor, the *zpu* processor in our experiments. This is not a problem if all processors used in

the architecture have sufficient performance. Accessing the common memory also incurs a performance cost as a result of the overheads involved with mbox indirection.

Different types of interpreted programs will require different numbers of mbox calls. Highly algorithmic programs would spend more time executing native calculations and would see less of a performance hit compared with memory access intensive programs.

The benchmark was to find all the prime numbers less than 5000 and was run on three architectural variants: a single *zpu* processor; a set containing all three different processors without any context switching, and the full architecture of four sets of all three processors. The results are shown in Figure 5.

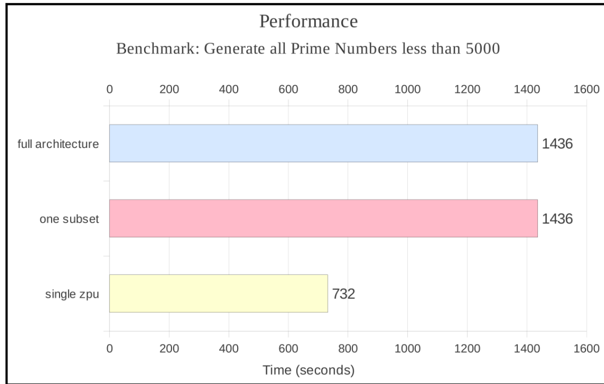

Figure 5: Performance benchmarking of the software interpreter.

The overhead of the mbox accesses increases run time for the example program from 732 seconds to 1436 seconds, this equates to a performance decrease of around 49%. However, even this kind of performance decrease would be acceptable in many safety or security critical applications. The addition of fragmentation through context switches adds no discernible run-time to the application. This is due to the context switch requiring no explicit state saving or restoring to occur.

Also of interest is the total number of read and write accesses through the mbox interface for the benchmark; this information is shown in Table 1.

Lines of Interpreted Code	264697
Number of *mbox* Read Accesses	42678143
Number of *mbox* Write Accesses	10278655
Avg. *mbox* Read Accesses per line	161.23
Avg. *mbox* Write Accesses per line	38.83

Table 1: Analysis of *mbox* accesses.

The *zpu* implemented in our experiment is a very poor performing processor. With the *zpu* executing natively at 25MHz, the average mbox access takes approximately 330 clock cycles. A typical `_mbox_read()` call as shown in Listing 3 expands to over 70 instructions on the *zpu*, which are executed at between four and five clock cycles per instruction.

The `_mbox_read()` call comprises four fixed mbox register writes (three to `mbox_ctrl`, one to `mbox_addr`), one fixed mbox register read (`mbox_datai`), and potentially multiple reads from the mbox status register (`mbox_sts`). In our experimental architecture once the address (`mbox_addr`) and control (`mbox_ctrl`) registers have been written it takes 3 clock cycles for the TCB to return the data, hence for the slowest performing *zpu* processor it will only need to read the status register (`mbox_sts`) once.

```
unsigned long _mbox_read(unsigned long addr)
{
  unsigned long returnValue;

  *mbox_addr = addr;
  *mbox_ctrl = 0x0;
  *mbox_ctrl = MBOX_READ | MBOX_ENABLE;

  while ((*mbox_sts & MBOX_RDY) != MBOX_RDY);

  returnValue = *mbox_datai;
  *mbox_ctrl = 0x0;

  return(returnValue);
}
```

Listing 3: `_mbox_read()` call.

Increasing the amount of data and state stored in common memory, and hence the required number of mbox accesses, facilitates easy fragmentation and better restricts an individual processor's access to program code and data. Decreasing what is stored in the common memory improves native performance, at the expense of more complex software support for fragmentation and lower data confidentiality.

No effort was placed into optimising the *ubasic* interpreter code to reduce the number of mbox calls, or increase the general efficiency of the program. Opportunity exists to perform multiple computations with global variables without having to read and write them back to the common memory, thereby reducing the number of mbox calls, and increasing performance.

4.1.2 Trojan Resistance

The example program fragmented the generation of prime numbers, with the TCB randomly selecting a new subset of processors to calculate every new prime number. On average, each processor only had access to one in every four prime numbers. This approach demonstrates that as the number of processors is increased, and with the use of judicious context switching, this architecture is capable of successfully partitioning sensitive information across different processors.

We developed several other BASIC programs to run on our architecture. These included a *pi* estimation program and a simple "access" type program that asked for a password and granted or denied access based on whether a hash of the supplied password matched a stored hash value. Context switches occurred after each character was read from the keyboard and the generated hash updated. Again, in a simplistic manner this demonstrates how individual processors, and hence any associated Hardware Trojans might be prevented from having access to sensitive data in its entirety.

Using the common memory to store shared program code and data enables execution state to be switched between different sets of processors. The protection against data leakage then depends on the frequency and granularity of fragmentation. Performance is affected by the amount of data accessed through the common memory and there is a trade-off between security and efficiency. This trade-off is able to be managed by the software developer. The mbox architecture features can be used to protect only the most important data structures, thereby minimising the performance hit, or they can be broadly used as is the case for the software interpreter.

The replication of execution prevents any minority set of processors from modifying the behaviour of the program (either outputs or execution of the BASIC program) or from leaking data through a logical system interface. A unanimous voting mechanism was implemented in the TCB for our experiments. Whenever any one of the three processors in the currently active subset was reset, or attempted an incorrect (unordered) mbox access, the system halted. For protection against DoS attacks, and functional and behavioural modification, different voting mechanisms would need to be prototyped.

4.1.3 TCB Analysis

The TCB for this architecture consists of a multiplexer/demultiplexer and voting arrangement operating on the mbox interfaces. The architecture of the TCB exhibits similar properties to that of SAFER PATH. Table 4.1.3 compares synthesised resource usage within the Xilinx Virtex 6 FPGA for a minimal processor core (Leon3) against the TCB for one ($b = 1$) and four ($b = 4$) subsets (or banks) respectively. Each subset contains 3 different processors.

	LUT6s	Registers
Single Leon3 core	2516	1221
TCB, 3 PEs (b=1)	125	334
TCB, 12 PEs (b=4)	557	1158
TCB[2], 3 PEs (b=1)	213	71
TCB[2], 12 PEs (b=4)	708	76

[1] Results obtained using Xilinx ISE Release 14.2
[2] SAFER PATH TCB

Table 2: TCB size analysis.

The TCB logic remains smaller than a single processor up to a threshold number of processors. The TCB is made up of simple, replicated circuitry that can more easily be checked for correctness than for example a CPU Arithmetic Logic Unit (ALU). As the number of processors increases, the TCB scales linearly due to the increasing size of multiplexers and demultiplexers. In contrast to SAFER PATH, the current design implementation registers external memory and message box inputs and outputs thus resulting in a high register count. With further optimisation, register usage could be reduced with minimal impacts on design performance.

The mboxes have not been optimised for performance, however they could be tailored either for specific applications or processor architectures. Mboxes with deep data registers and increased voted block size could also be considered.

4.2 VNC Client

The software interpreter shows how the architecture can be used, and even abstracted from a software point of view. This second experiment was performed to demonstrate how a larger, more complex application could be ported to the architecture. The rationale behind choosing a VNC Client is to provide a simple thin-client, where Hardware Trojans residing within the processing elements could not compromise the session.

A VNC client was implemented on a version of the architecture that included only three processors. To support this, a serial port, PS2 keyboard, PS2 mouse, and framebuffer memory were mapped into the common memory and IO space, with access supervised by the TCB. The VNC client communicates with a server through the serial interface, reads in keyboard and mouse events through the PS2 interfaces, and writes to a display via a double-buffered 800x600 framebuffer memory. The mbox connected peripheral hardware is shown in Figure 6.

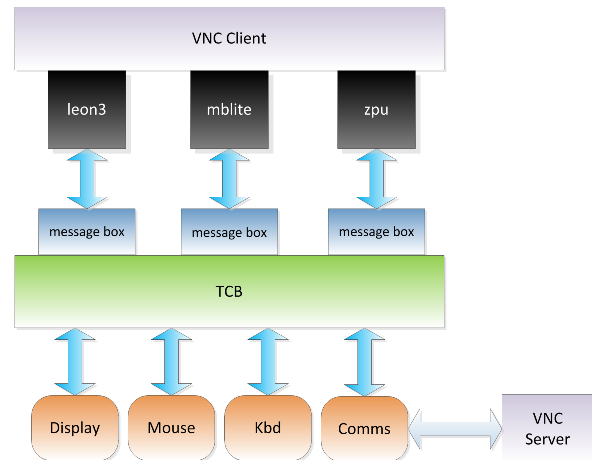

Figure 6: VNC client architecture.

The *fbvnc* (Weidner 2000) VNC client was modified and ported to each of our three implemented architectures (*leon3*, *mblite* and *zpu*). Minor architecture-specific code differences are required and the generated machine code is vastly different for each processor, but the ordering of mbox accesses to the common memory space is maintained. For example, as the program executes on each architecture, each processor reads from the keyboard, mouse or serial ports, and writes to the display or serial ports in the same order.

The VNC client communicates over a 1Mbps serial link, proxied via a network connection to a VNC server. Using hextile encoding, acceptable performance is obtained running the processors at 100MHz.

In arranging our architecture in this manner, we are able to ensure that each VNC client is provided with identical inputs, and that those inputs generate identical outputs. The synchronised execution and voting provides protection from malicious modification via the untrusted processors. However, in this instance, sensitive information that is being processed by the VNC client may be able to be leaked by an infected processor. The VNC communications protocol is modular, so fragmentation could be added to improve data confidentiality. Access to keyboard and mouse inputs, and data destined for the framebuffer would then be limited to small windows for each processor, helping to mitigate the damage of any data leakage.

The VNC client shows a how a more complex application can be implemented. Here the architecture is usefully applied to protect the inputs and outputs of a system from Hardware Trojan interference.

4.3 Summary

The two demonstrators show how applications can be protected against the threats of Hardware Trojans.

The software interpreter maintains all the protection properties of the earlier SAFER PATH architecture; a single program code, in this instance a BASIC program, can have its execution replicated and fragmented over many different processors. Sufficient

performance remains in the architecture for successful application within a security critical system, while the TCB remains simple enough for accreditation.

Interpreting a program is inherently slower than native execution; this is true for all platforms. However, interpretation brings us the benefits of programming language abstraction and allows us to write a program once and run it anywhere. This is especially true for this architecture where the overheads of writing a custom application are high. The utility of a generic interpreter was demonstrated when newly written BASIC programs were able to immediately take advantage of our architecture's replication and fragmentation properties, with minimal to no work required of the programmer.

Although the use of the *ubasic* interpreter has merit for our experimentation, a different interpreter, for example a Java Virtual Machine (JVM), that has a more efficient byte code representation and better execution efficiency may provide improved performance. This improved performance comes at the cost of a larger initial effort to port the code to multiple architectures and to add support for fragmentation. The complexity of the TCB also increases if support is required for real-time features, e.g., timers and interrupts. However, a JVM would also provide the opportunity to access the large existing Java byte-code application base.

Increasing the barrier for successful Hardware Trojan operation forces Hardware Trojans to become more complex, usually translating into a larger implementation footprint. This makes them more easily detected through current Hardware Trojan detection mechanisms.

5 Further work

Our experimentation ran native applications on bare metal processors. The architecture works equally as well for more complex processors running multi-threaded operating systems. This holds as long as strict ordering is maintained through a dedicated mbox interface for any specific application that is to be protected on the architecture. Hence protected applications can run along-side less trustworthy applications on the same processor. The multi-threading nature of the underlying operating system also ensures a processor can still be usefully occupied while blocking on mbox accesses of the protected application. Further, whilst our experimentation was focused around FPGA development, the architecture is not limited to FPGA instantiation. Discrete processors could be combined, either at a macro level or together on a PCB like substrate to form a Hardware Trojan resistant computing platform.

Improving application design, mbox design and link speed, or enabling concurrent use of all available PEs could improve performance. Mboxes could be extended to distribute interrupts via register style interfaces, with consideration given to the impact on synchronised execution.

Algorithms for tuning fragmentation to achieve optimal data confidentiality properties should be investigated. These may be instrumented through the software build process or by source to source transforms enabled through formal methods.

6 Conclusion

The architecture presented allows computation to be replicated and fragmented across a pool of widely heterogeneous processors. Unlike SAFER PATH, there is no longer a requirement to obtain variants in manufacturing or design. Our updated architecture can be implemented using entirely COTS processors.

A minimal TCB, amenable to accreditation, votes on loosely synchronised, but replicated behaviour. This collective behaviour is probabilistically correct, providing integrity and availability in the presence of active Hardware Trojans. Further, fragmenting this behaviour limits individual processor access to data and defends against data leakage attacks.

A prototype implementation within an FPGA and two software applications were developed to demonstrate the utility of the architecture. The first was a software interpreter executing arbitrary programs, with an acceptable performance decrease for intended security critical applications. Extending the software interpreter from BASIC to a more sophisticated platform, such as a JVM would dramatically increase the utility of the system. The second application was designed to protect a VNC session executing on a thin client with untrusted COTS processors.

These applications demonstrate the use of the architecture as a replacement for an embedded or desktop processor, especially in circumstances where system operation needs to be guaranteed, or where sensitive data is being processed.

References

Abramovici, M. & Bradley, P. (2009), Integrated Circuit Security: New Threats and Solutions, *in* 'Workshop on Cyber Security and Information Intelligence Research', CSIIRW'09, ACM, New York, NY, USA, pp. 55:1–55:3.

Aeroflex Gaisler AB (2010), 'Leon3 Multiprocessing CPU Core Product Sheet'. http://www.gaisler.com /doc/leon3_product_sheet.pdf.

Baumgarten, A., Steffen, M., Clausman, M. & Zambreno, J. (2011), 'A case study in hardware trojan design and implementation', *Int. J. Inf. Secur.* **10**, 1–14.

Baumgarten, A., Tyagi, A. & Zambreno, J. (2010), 'Preventing IC Piracy Using Reconfigurable Logic Barriers', *IEEE Des. Test. Comput.* **27**(1), 66–75.

Beaumont, M., Hopkins, B. & Newby, T. (2012), SAFER PATH: Security Architecture using Fragmented Execution and Replication for Protection Against Trojaned Hardware, *in* 'Design Automation and Test in Europe (DATE)', pp. 1000 –1005.

Bloom, G., Narahari, B., Simha, R. & Zambreno, J. (2009), 'Providing secure execution environments with a last line of defense against Trojan circuit attacks', *Computers & Security* **28**(7), 660 – 669.

Chakraborty, R., Narasimhan, S. & Bhunia, S. (2009), Hardware trojan: Threats and emerging solutions, *in* 'IEEE High Level Design Validation and Test Workshop', pp. 166 –171.

Collins, R. R. (1998), 'The Pentium F00F Bug'. accessed at http://www.rcollins.org/ddj/May98/F00 FBug.html, 19 October 2012.

Dunkels, A. (2007), 'uBASIC - A really tiny BASIC interpreter'. accessed at http://www.sics.se/~adam /ubasic, 24 November 2011.

Jin, Y., Kupp, N. & Makris, Y. (2009), Experiences in hardware trojan design and implementation, *in* 'Hardware-Oriented Security and Trust,

2009. HOST '09. IEEE International Workshop on', pp. 50 –57.

King, S. T., Tucek, J., Cozzie, A., Grier, C., Jiang, W. & Zhou, Y. (2008), Designing and implementing malicious hardware, *in* 'Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats', USENIX Association, Berkeley, CA, USA, pp. 5:1–5:8.

Kranenburg, T. & van Leuken, R. (2010), Mb-lite: A robust, light-weight soft-core implementation of the microblaze architecture, *in* 'Design, Automation Test in Europe Conference Exhibition (DATE), 2010', pp. 997 –1000.

Lin, L., Burleson, W. & Paar, C. (2009), MOLES: Malicious Off-Chip Leakage Enabled by Side-Channels, *in* 'Proceedings of the 2009 International Conference on Computer-Aided Design', ICCAD '09, ACM, New York, NY, USA, pp. 117–122.

Lin, L., Kasper, M., Paar, C. & Burleson, W. (2009), Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering, *in* 'In Cryptographic Hardware and Embedded Systems - CHES 2009, volume 5747 of LNCS', Springer, pp. 382–395.

McIntyre, D., Wolff, F., Papachristou, C. & Bhunia, S. (2010), Trustworthy Computing in a Multi-Core System Using Distributed Scheduling, *in* 'On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International', pp. 211 –213.

Rajendran, J., Gavas, E., Jimenez, J., Padman, V. & Karri, R. (2010), Towards a comprehensive and systematic classification of hardware trojans, *in* 'Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on', pp. 1871 – 1874.

Reis, G., Chang, J., Vachharajani, N., Rangan, R. & August, D. (2005), SWIFT: Software Implemented Fault Tolerance, *in* 'Code Generation and Optimization, 2005. CGO 2005. International Symposium on', pp. 243 – 254.

Saxena, N. & McCluskey, E. (1998), Dependable Adaptive Computing Systems- The ROAR Project, *in* 'Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on', Vol. 3, pp. 2172 –2177 vol.3.

Tsang, R. (2009), Cyberthreats, Vulnerabilities and Attacks on SCADA Networks. University of California, Goldman School of Public Policy, working paper, accessed 19 December 2011, http://gspp.berkeley.edu/iths/Tsang_SCADA%20 Attacks.pdf.

Waksman, A. & Sethumadhavan, S. (2011), Silencing Hardware Backdoors, *in* 'Proceedings of the 32nd IEEE Symposium on Security and Privacy, May 2011'.

Weidner, K. (2000), 'fbvnc - a framebuffer-based VNC client'. accessed at http://pocketworkstation.org/fbvnc.html, 15 December 2011.

Yeh, Y. (1996), Triple-Triple Redundant 777 Primary Flight Computer, *in* 'Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE', Vol. 1, pp. 293 –307 vol.1.

Young, D. (2011), 'COTS technologies ready for UAV deployment', *Military Embedded Systems* **7**, 12.

Zylin Consulting (2008), 'Zylin CPU'. http://opensource.zylin.com/zpu.htm.