A System-level Security Approach for Heterogeneous MPSoCs

Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic
Department of Electrical and Computer Engineering
University of Auckland, New Zealand
ptan262@aucklanduni.ac.nz, {m.abhari, z.salcic}@auckland.ac.nz

***Abstract*—Embedded systems are becoming increasingly complex as designers integrate different functionalities into a single application for execution on heterogeneous hardware platforms. In this work we propose a system-level security approach in order to provide isolation of tasks without the need to trust a central authority at run-time. We discuss security requirements that can be found in complex embedded systems that use heterogeneous execution platforms, and by regulating memory access we create mechanisms that allow safe use of shared IP with direct memory access, as well as shared libraries. We also present a prototype Isolation Unit that checks memory transactions and allows for dynamic configuration of permissions.**

Keywords—embedded systems; security; access control; multiprocessor; isolation

I. INTRODUCTION

Embedded systems are becoming increasingly sophisticated as designers consolidate different functionalities into a single application, mixing together sensitive data and physical actuation, and executing such applications on highly integrated hardware platforms. These complex systems feature various timing requirements, as well as various levels of security requirements. The notion of mixed-criticality, in different aspects like timing, and security, becomes inherent in their designs [1]. To accommodate application complexity there is a trend towards the combination of different processors, hardware Intellectual Property (IP) blocks, and shared memory resources into a heterogeneous multiprocessor system on chip (MPSoC). As these systems become increasingly exposed through their connections with different networks [2], security becomes an important consideration.

In traditional uniprocessor platforms the approach to system security often relies on a central authority, such as an operating system (OS) [3], to manage access to system resources, and to take advantage of security features in the platform architecture, such as memory protection units (MPUs). There is a hierarchy of privilege, where the OS has access to *all* elements of the hardware platform. Application tasks in the system run at a lower level of privilege, and are restricted by permissions that are set by the privileged OS.

However, in embedded systems that use MPSoCs the approach needs to be rethought. It is often challenging to deploy a single OS across the platform, particularly when some parts run "bare-metal" to meet timing requirements. The hardware platform may also have IP blocks that have direct access to shared memory. We may also need strong guarantees that parts of the application are completely isolated. An OS or other component which has the ability to access all parts of platform provides a lucrative target.

In this paper we explore a system-level approach to security, which requires a rethinking of application design alongside additions to the platform architecture. We focus primarily on the idea of isolation and access control as the *security foundation*, and argue for the need to avoid a central trusted authority at run-time for managing security. We present a strategy which distributes authority for managing access permissions to application tasks at run-time, and we investigate the implications of adding hardware support for this strategy in the form of Isolation Units (IUs), which check memory accesses and allow tasks to manage their own permissions. Our contributions are:

- A discussion of security requirements in complex embedded systems, based on an example motivating scenario in the domain of home automation
- An examination of access control policies in the context of an example embedded system design on a heterogeneous hardware platform
- A system-level security approach based on the notion that tasks themselves are responsible for managing access to their own regions of the address space, based on ideas from discretionary and role-based access control [4, 5]. This decouples any OS from being primarily responsible for security
- A description of mechanisms using our security approach for supporting shared libraries and shared IP with direct memory access capabilities
- Prototype IUs that can be managed and reconfigured at runtime, without using an OS, with permissions transferred between tasks, as well as to IPs.

In Section II we describe a motivating scenario for security in MPSoCs, as well as the threat model that we address. Section III explores various access control policies, and draws from the motivating scenario to determine the security requirements. Section IV proposes our system-level approach based on permissions, and discusses the necessary architectural support for this approach. Section V presents an IU prototype with the discussion of the synthesis results targeting an FPGA in Section VI. In Section VII we discuss our approach in the context of related work and conclude in Section VIII.

II. Motivating Scenario

A. The Integrated Home Automation Hub

Consider an example embedded system, the Integrated Home Automation Hub, which integrates different functions in a home: environmental control, home security, fire detection, and entertainment. This embedded system is implemented as the Hub application, where control of each function is partitioned into number of tasks, as shown in Table 1. We define a task as a unit of software with its own program code that can execute concurrently with other tasks. They can interact with tasks from their own group, or with those from other groups. For example, an owner returning home might unlock her door with her fingerprint, turning on lights and music – this is an interaction between home security, environmental, and entertainment task groups.

This Hub application runs on a heterogeneous MPSoC, like that shown in Figure 1, with an example task group allocation as indicated in Table 1. We first consider this as an unsecured platform. The MPSoC platform consists of different types of processors (P_X, P_Y, and P_Z), together with a mix of shared resources (memories, I/O, and hardware IP peripherals) which are accessed through a shared interconnect infrastructure. The processors vary in complexity, with P_X being highly complex (for high performance in multimedia functionality) and P_Z dedicated to a single task (e.g. used for time-predictability). P_Y lies somewhere in between. Some of the IP in this platform have direct memory access capabilities; for example, one could be an accelerator for signal processing, and another could be a graphics controller for displaying information/media on a local panel. We map I/O, IPs, and memories into a global-shared address space; the shared interconnect provides the ability for tasks and IP blocks to perform memory accesses to any part of the platform. Some tasks run on "bare-metal" (P_Z), some might execute with hardware-based or static scheduling (P_Y), and others might employ an OS (P_X).

In this Hub application we can *attempt* to categorize our task groups by "criticality", distinguishing between task groups like fire detection (which has strong real-time requirements) and entertainment (which is more of a convenience function). We might consider a task group "more-critical" if it uses I/O peripherals for physical control, has real-time requirements, or has sensitive data that should be protected from malicious attack, while other "less-critical" task groups might use IPs that are not required by the "more-critical" task groups, or has less sensitive data. Alternatively, we might classify a task group which involves user interaction as "more-critical" from a security perspective, due to possible exposure to malicious entities. However, resources may be shared between task groups, with some peripherals and memory shared, as shown by the multi-coloured boxes in Figure 1. In fact, criticality classifications may be very complex, depending on the mix of criteria, or be subject to an overly simplistic view of our design if we simply specify "critical/non-critical", or "secure/non-secure".

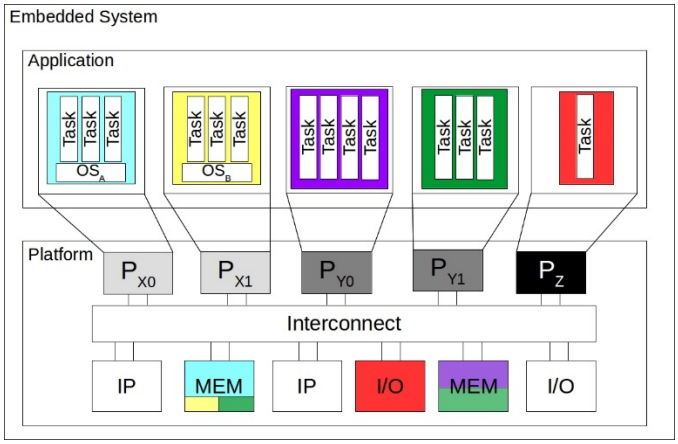

Figure 1: An embedded system, comprising of an application made up from task groups, and executing on a heterogeneous MPSoC platform. There are different types of processors, and different ways in which resources might need to be shared, indicated by the multi-coloured boxes.

Furthermore, we cannot completely isolate our task groups based on task group classifications alone. Consider tasks *within* task groups: the role of E2 is to process sensor data from I/O. R0 is used to display the sensor statuses, implying a relationship across some sort of "criticality" boundary. Moreover, tasks of similar "criticality" do not necessarily need to access the same resources – N0 plays audio to speakers and should not necessarily have access to other I/O, nor should R0, which handles remote user requests, have access to the signal processing accelerator used by the media decoder N1. Even within task groups we can have complications – the

Table 1: The Integrated Home Automation Hub – Hub Task Groups and Descriptions

Task Group (processor)	Tasks		Functional Description	"Criticality" Remarks
Environmental Control (P_{Y0})	E0	Light	Controls the lights	Physical control with human impact
	E1	AC	Controls the heating for each room	Physical control with human impact
	E2	Sensor	Receives and processes sensor data	Physical sensing, real-time requirements
	E3	Environment	Uses sensor data to manage environment	Physical control, real-time requirements
Home Security (P_{Y1})	S0	Security camera	Controls security cameras, processes footage	Sensitive information, real-time requirements
	S1	Door access	Controls door locks	Physical control, sensitive
	S2	Biometric access	Manage fingerprint entry system	Real-time requirements, sensitive data
Fire detection (P_Z)	F0	Fire manager	Detects fires, raises alarm, and extinguishes	Real-time requirements, must always be active
Entertainment (P_{X0})	N0	Speaker	Controls multi-room speaker system	Physical control, less-critical
	N1	Media decoder	Manages media files, and processes audio	Potentially vulnerable, non-sensitive data
	N2	Local interface	Manages local control panel for UHAH	User interface, potentially vulnerable
Remote management (P_{X1})	R0	Remote interface	Web interface for remote management	User interface, potentially vulnerable, exposed
	R1	Update	Manages remote update of controller software	Potentially accesses critical data
	R2	Authentication	Authenticates remote users (checks passwords)	Accesses sensitive data

authentication task R2 should have access to user passwords, implying a higher level of "criticality" in what might be considered a "less-critical" task group. We also have IP blocks that can perform memory accesses; these too should be classified, although this is challenging if they can be shared between different tasks.

In light of the complexities when it comes to "criticality" classification, as well as different tolerances to exploited vulnerabilities, we need to introduce a means to isolate tasks, and task groups, from each other. For example, we might be able to tolerate an attack on our entertainment functionality, but not if it results in extraction of our secret passwords.

This indicates that there is a need for *fine-grained* access control, and even *temporary* access permissions. For example, R0 is used to display the system status and also to update security camera software. This should *only* occur if the remote user is authenticated – R0's ability to access a critical part of memory should be on-demand and temporary. Here, R0 should communicate with R2 to authenticate the user; R0 then uses commands available in R1 to update software. However, R1 should not be able to access all program memory – it should only be able to modify what is *specifically* required.

Finally, we can also consider that the compromise of one of the tasks should not necessarily constitute complete system failure; other tasks should be able to run unaffected, if possible. In a system like the Hub it is desirable that environmental control is still usable and can be "trusted" even if the entertainment application is compromised.

B. Threat Model

The architecture of the MPSoC platform has two major sources of risk: the shared interconnect and shared resources. The shared interconnect facilitates access to all parts of the platform, and shared resources bridge task groups. As our resources are memory-mapped we consider this as a memory protection problem. In our threat model we assume that a task can be compromised so that it can perform arbitrary memory accesses. Other tasks are otherwise functionally correct, and physical attacks are considered to be outside the scope of this work. As a result of a task's compromise, we have the following potential attacks:

- **A1.** If the compromised task is running bare-metal it can generate a memory access transaction to any part of the platform. This can result in disruption of other tasks or access of sensitive data
- **A2.** If the compromised task can configure an IP with direct memory access capabilities, it can configure the peripheral to access a part of memory on its behalf.

If we use an OS, and processors with a built-in memory protection mechanism (such as an MPU), we can alleviate these problems to some extent. A "traditional" OS can configure the MPU to introduce a level of access control. However, if the task that has been compromised has the ability to escalate its privileges to that of the OS, additional attacks include:

- **A3.** Any MPU is disabled, thus enabling unprotected access to the rest of the platform.
- **A4.** The MPU is reconfigured to disrupt the operation of other tasks located on that processor, i.e. by restricting memory accesses such that the other tasks cannot perform their intended functionality.

There are many ways in which a task can become compromised [6]; for example, one attack vector could be code injection via a buffer overflow in R0. In the absence of any security features the attacked task can access parts of the platform it would not ordinarily have access to, such as disabling fire protection by manipulating parts of memory. As an example of A2-type attack, consider N2 (this task manages a control panel). If it is compromised, N2 could be commandeered to program a graphics controller used for display so it stealthily probes and collects sensitive data from the memory of other tasks. An example of this type of attack is explored in [7].

The security approach we propose in the next section aims to alleviate the impact that a compromised task may have on other parts of the overall system.

III. Security Model Discussion

A. Examining Access Control Policies

The problems outlined in the previous section largely arise from the lack of proper memory access control. Hence, we first consider the formulation of a suitable policy, drawing from our motivating scenario for requirements that may exist in an embedded system.

At a most basic level we need to place restrictions on the tasks in the system, constraining their ability to read and write to parts of memory. As discussed in [5] there are several different types of access control policy that can be implemented, each with their own merits and drawbacks.

The most restrictive of these is the classical Mandatory Access Control policy (MAC), which relies on a centralized and privileged administrator to manage the access permissions for all tasks in the system. At run-time, this is usually a job performed by an OS kernel, as in (SE) Linux [3].

A more flexible policy is based on the idea of Discretionary Access Control (DAC), where entities in a system are able to grant and revoke access of objects to each other. Another alternative is Role-based Access Control (RBAC) [4] where permissions are attached not to tasks directly, but to roles, where a role is defined as a specific "job" or "function" that a task may be performing. A task might be allocated different roles at run time, thus receiving the associated permissions. A mixture of these policy styles can be used as a starting point for our overall security policy.

In order to decide what functions we need to support with our security model, we look again at the Hub example:

1) Shared Data: As an example, E4 (environment control) should have access to processed sensor data, which should be provided by E2 (sensor task). Rather than provide E4 access to the sensors directly, we want E4 to have access *only* to the data it requires (in accordance with the Principle of Least Privilege [8]). As a result, our security model should allow controlled access to a region of memory that is accessible between the two tasks. Additionally, this relationship implies

that E4 should be provided only with *read* access, while E2 requires at least *write* access. Another scenario: a local user uses the local interface (N2) to access and review security camera footage from S0. N2 first needs to make sure that the local user has rights to access the footage; it will need to communicate with S2 (biometrics) to authenticate the user, and only then should the security footage, managed by S0, become accessible. N2 should not *always* have access to the security footage as this constitutes a security risk (after all, N2 interacts with the external world). This indicates that we require a mechanism for *dynamic* permission changes.

2) Shared Code: Consider N1 and N2, these two tasks share some common code, such as a library for displaying contents from a media file. Such libraries are similar to IP blocks, in the sense that they are resources which provide a "service" for different tasks. Shared code has the potential for being exploited by malicious tasks to "extend" its influence. We need a mechanism to allow shared libraries to perform their job on only the parts of memory that they are immediately concerned with at any point in time.

3) Shared IP blocks: IP block sharing can be considered as a mixture of 1) and 2), where we share data between tasks and the IP, while also making use of extra functionality. N1 (multimedia) and S2 might both use the same signal processing accelerator. While the Hub application executes, the accelerator needs access to data that belongs to N1 and S2, but *not necessarily at the same time*. Our security approach should prevent data leakage, such as the case where N1 can attempt to gain access to the data of S2 by configuring the accelerator. This scenario implies the need to be able to perform some manipulation of permissions at run-time, where the accelerator can access only the data it has been provided with at that moment in time. In other words, the IP block plays a different role at different times throughout the application's execution. In fact, our approach needs to consider IP blocks similarly to software tasks, in terms of permissions. Managing the sharing of peripherals can be handled in the usual manner, such as through use of mutex objects.

B. Discussion

In our approach we are trying to formulate a policy that can encompass the *entire* system. From the informal analysis, we can generalise the required abilities to these two key features:

1. The ability to set up regions where two (or more) tasks/IP blocks can share data, with a mixture of read/write permissions.
2. The ability to grant/revoke permissions at run-time.

A MAC-based policy would need to introduce a central authority that is invoked every time one of the dynamic permission changes occurs. However, tasks and task groups in the Hub application do not easily map to "security levels", as suggested in [5], as there is no clear hierarchy in terms of data sensitivity. Sharing IP blocks and code lends itself well to RBAC, as IP/code can be seen to perform a different "role" depending on the task it is working for. Tasks also intuitively "own" data, passing it on to other tasks or IP blocks for further use at run-time. For example, a task that manages sensors first collects and processes sensor data, and then passes it to another task to indicate the status of the sensed environment. In this case, a DAC policy seems quite useful too.

It appears that a mixture of these policies would be most useful; all tasks and IP blocks in the system need to have a set of permissions, some of which are temporary. Fixed permissions can be set, as in a MAC policy, by a trusted authority (the designer) *at design time*. However, if we wish to support more dynamic permissions at run-time without a centralised authority (and the potential risk involved with that), we need to consider a different, more decentralized approach, which we now discuss in the following section.

IV. Proposed Approach

A. Permissions

The alternative to having a single centralized authority in the system is to have, *multiple* authorities. In other words, we delegate and distribute responsibility for managing permissions in the system. While this shifts complexity into tasks themselves we can better isolate systems should there be a successful compromise, as there is no entity that has default access to all regions. Starting with a partitioned global shared address space as the foundation, we make application tasks the owners of their own parts of the address space. Tasks bear responsibility for managing access. To do so we present three types of permissions:

1. *Base* permissions are statically allocated by the designer, but cannot be transferred to anybody else. They indicate regions that are accessible to a task/IP block.
2. *Owner* permissions are statically allocated, and indicate regions that a task has authority to share. It is not necessary that all tasks "own" at least one region
3. *Shared* permissions are dynamic and are used to grant access to regions to other tasks; the owner can revoke access at any time.

As shown in Figure 2, the designer is the primary authority, determining the regions each task/IP block can access during the design phase. Static permissions are always active. Once the system runs, tasks which own regions can then create shared permissions for other entities; for example, Task A allocates a subset of the region it owns and gives the

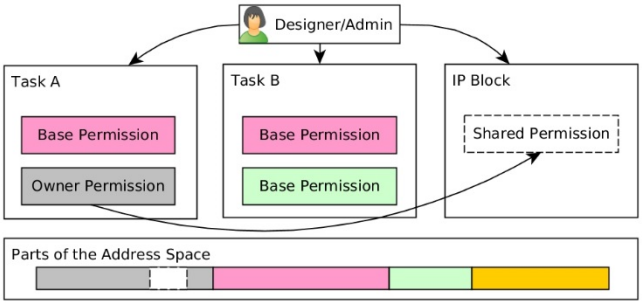

Figure 2: Types of permissions

IP block access to that region. One implication of this approach is that the entire address space (including memory-mapped peripherals) should be allocated in advance, with owners designated and base permissions set. This approach provides a foundation for a more secure system with tasks completely isolated if there is no interaction between them.

1) Supporting IP block and Code Sharing: We use a mixture of all three types of permissions to ensure that resources are shared only when required, and that isolation boundaries are respected. Firstly, let us consider IP blocks. They typically have a set of memory-mapped registers used to configure that particular device. We call these registers collectively a *Service Request Region (SRR)*. Shared code is very similar, in that a task might "request" a certain function be executed, specifying the location for the output. IP blocks and shared code are a type of *shared resouce*.

If a task needs to use one of these shared resources, we can allocate a (write-only) base permission to that task to access the SSR of that resource. The resource itself will have no permissions to any part of the system, aside from a base permission to its own private working memory if needed.

If a task wishes to use the resource, the task first acquires control of that resource (such as by gaining a mutex lock). It sets up the data it requires the resource to process, as well as a region of its memory where it wants the result to be stored. It then writes its request (i.e. configuration) to the SRR of that resource. It then sets up a shared permission for that resource, granting access to the data regions so it can do the job. On completion, the task then revokes the shared permission.

2) Shared data: A task shares data by granting other tasks access to that region using a shared permission if the shared memory region is temporary. In order to facilitate run-time requests for data sharing, tasks can have base permissions that are allocated at design time for fixed, always-accessible communication buffers in shared memory.

B. Security Implications

The described approach allows us to capture the relationships between tasks and resources, and by careful design we can completely isolate systems that do not interact. Additionally, because we distribute authority for granting/revoking permissions throughout the system, we can limit the extent to which a compromised task can impact the system as a whole. However, this approach does not in itself constitute a completely secure system; tasks could still be compromised, and these could still have *some* impact. On the other hand, if we are able to detect violations, our dynamic permissions allow us to respond with strategies like quarantine of compromised tasks, at least in terms of revoking shared permissions, thus allowing for continued (although possibly degraded) system operation.

We can extend this security approach in several different ways. For example, we can split tasks up into a 'worker' and 'supporter' to detect when memory regions are overflowed. The 'supporter' creates and shares memory regions for the 'worker'; if an overflow occurs the attempt to access memory results in a permission violation.

C. Architectural Support

In order for our security approach to be effective we need to guarantee that *all* parts of the overall system are subject to the same security enforcement. As such, there are three key operations that we need to support: enforcement of permissions, identification of the running task, and management of permissions. In this subsection we discuss some potential architectural options for these operations.

1) Enforcement: Every memory transaction in the system needs to be checked against a list of permissions. There are two main aspects to consider here: permission storage, and transaction checking. Ideally, all memory transactions are checked, which indicates that a protection unit (such as an MPU) should be placed at the memory interface of every component that can access memory. Intuitively, this will result in both resource and performance overheads. As discussed in [9], this cost can be reduced by checking transactions on cache misses, if caches are available, thus reducing checking frequency. However, this introduces the possibility for information leakage: multiple tasks could share a processor, and sensitive data from one task could remain resident in the cache while another task runs. More overhead is introduced depending on whether permissions are stored centrally or are distributed; if there are multiple permissions for each task there may be a need to "fetch" these from elsewhere in the platform. Further consideration might also be required if tasks are able to migrate across processors, but this is out of the scope of this paper.

2) Task Identification: To properly enforce permissions we need to correctly identify *the running task*; this is crucial as tasks could otherwise masquerade as other tasks to circumvent isolation boundaries. Accesses that fall outside permitted regions also needs to be blocked and identified for handling. In traditional systems, tasks are identified by the OS which sets a task ID register when tasks are loaded for execution. However, in the context of untrustworthy software there is a danger in relying on software-based identification. One mechansim could be to identify tasks by noting the processor that issues a memory transaction. Every processor and IP block is given an ID number and memory transactions are checked at their physical origin, permissions are enforced at this level. This coarse-grained identification is most suited to situations where a single task runs on each processor, or where all tasks running on a processor have identical security requirements.

However, in our heterogeneous MPSoC it may be more appropriate to identify tasks within processors, with permissions enforced at this higher level of granularity. Instead of using an OS to identify tasks a hardware-based task loader could be used; this hardware performs task switching independently to the executing software by directly controlling the processor state and internal registers. This comes at the cost of invasive modification to existing processors, or creation of new architectures, but is immune from software-based exploits.

A higher granularity mechanism is available in the form of so-called Program Counter Based Access Control, or PCBAC, presented in [10], and used in some recent work [11]. Here task identity is associated with the address of program instructions. This has a fairly high level of trust as PC values are typically not able to be directly manipulated (aside from jump and branch instructions). In the event the PC is changed to one that corresponds to another task, the task ID would also change, applying a different set of permissions and thus complicating attack. In this case, shared code would need to be associated with a specific task, or marked as a task itself; our approach of setting up shared permissions at run-time nicely complements this approach.

In order to use this technique, access to the PC for each processor is necessary. If the internal architecture does not expose the PC this technique can be approximated by examining the address lines of the instruction bus; the confidence in task identification is reduced when this information is only available at lower levels of the memory hierarchy (e.g. the situation where only the connection between L2 cache and external memory is accessible instead of the connection between the processor and L1 cache).

3) Managing Permissions: In many processors where memory protection is directly supported special instructions are used to manage access control when running in "supervisor" mode. In order for our approach to work, we need an alternative to these special instructions, especially when a processor or IP does not have built-in support for privilege levels. Despite IP and processors being heterogenous, they are all able to perform memory accesses. By mapping permission management into the address space of each task we can allow tasks themselves to reconfigure their shared permissions at run-time. As permissions are granted/revoked between different tasks on different processors, and between IP blocks, these sets of permissions need to be connected together.

V. IU Operation and Implementation

To investigate the feasibility of our proposed security approach we designed an early prototype Isolation Unit (IU) and implemented it in VHDL. These IUs check permissions and allow permissions to be managed at run-time. IUs are added between each processor/IP block, and the interconnect, as shown in Figure 3. Depending on the nature of the processor/IP block connected, the complexity of the IU can vary. First, we use the term "local" to refer to the processor/IP block directly attached to the IU, and "remote" to refer to a processor/IP block elsewhere in the platform. In this work we

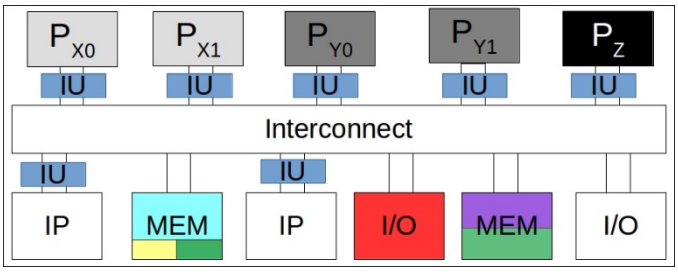

Figure 3: MPSoC with Isolation Units added

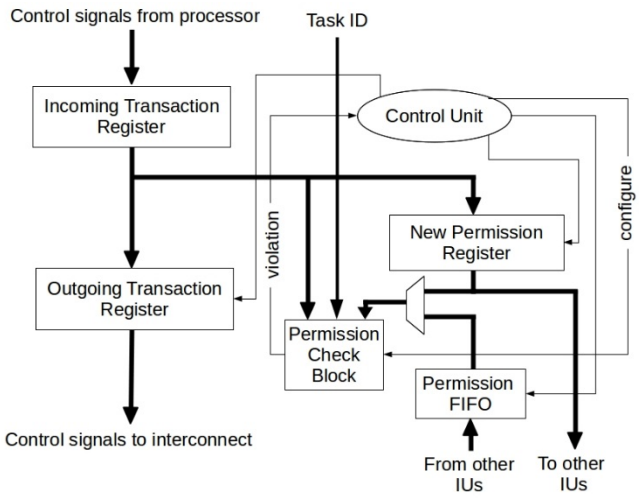

Figure 4: Simplified IU Internal Design

explore four different types of IUs:

1. an IU that enforces base permissions only – for use with processors that have no dynamic relationships between tasks
2. an IU that enforces base permissions and can support local reconfiguration – for use with processors that have several local tasks that have shared permissions
3. an IU that enforces base permissions and can support local and remote reconfiguration – for use with processors which have several tasks that have shared permissions, including with "remote" tasks and IPs
4. an IU that enforces only shared permissions – for use with IPs that receive permissions only

A. IU structure

Figure 4 shows a simplified view of an IU that supports both local and remote reconfiguration. All IU-types contain a register that holds the incoming transaction from the processor/IP block, a register that holds the transaction that is released to the interconnect, as well as a Permission Check Block.

The Permission Check Block contains registers for every permission that corresponds to the local processor/IP block, including registers for shared permissions, which are initially disabled. The Permission Check Block receives information from the incoming transaction register (nominally, the target address, and read/write signals). All permission registers are checked for access rights in parallel.

The IU also has a Control Unit for managing the modification of shared permissions, and a register for forming the new permission. A FIFO is used to receive new permissions from other IUs. If remote reconfiguration is not needed, the Permission FIFO is removed. If the IU does not need to be reconfigured at all, we remove the Control Unit completely.

The ID of the current running task is passed to the IU from either the local processor (if we use a software-based ID approach), a HW-based identifier (if we use a HW-scheduler, or task identification as implemented using the approach

described in [11]), or hard-coded (if we simply identify the IP block, or if there is a single task allocated on the processor).

The IU is memory-mapped into the address space of the local processor for reconfiguration; when a task writes to the IU's addresses, the New Permission Register stores the base address, end address, and whether or not the region is read and/or write enabled.

B. Permissions

Each permission consists of 74 bits: 32 bits used for the upper and lower region boundaries, 4 bits for the owner of the region (2 bits for processor, 2 bits for the task ID), 2 bits for the task ID of the task that has received the permission, and 1 bit each for whether the permission is for read and/or write access, whether the task owns the region, and whether the permission is valid.

C. Connecting IUs

We designed our IUs based on the idea that we directly connect each IU to all other IUs in the system – each connection contains a permission bus and a request signal for updating permissions. The permission bus connects to a FIFO buffer on the receiving end, for storing the incoming permission (our prototype has buffers that can hold four permissions per connection). When a task reconfigures a remote permission the IU places the permission on the required IU-pair permission bus, and asserts the request signal. On the destination IU the new permission is then added into the buffer. Provided that a local task is not setting up a new permission, the IU checks one of the IU-pair buffers; if the buffer is not empty the pending permission is read, and passed into the Permission Check block. In each cycle the IU checks a different IU-pair buffer, ensuring that all new permissions are eventually added.

D. Operation

1) Normal operation: In each clock cycle memory accesses from the local processor are captured in the Incoming Transaction Register; the permission checking occurs, and the transaction is released to the system interconnect if no violation is detected. This operation adds a *fixed single cycle delay* to all transactions. In the event a violation is detected (when the access is unauthorized, or the current task is invalid) the offensive transaction is blocked. A violation interrupt signal is asserted, which can be used to interrupt the processor or to notify another part of the system.

2) Reconfiguration procedure: **Step 1:** The task writes the base address of the shared region to the New Permission Register; the control unit redirects the write data into the Permission Check block to check that the task is sharing a region that it owns**. Step 2:** The task then writes the end address of the shared region to the second reconfiguration register. **Step 3**: To complete reconfiguration the task writes a word with the target task ID and desired permissions to the final reconfiguration address. **Step 4:** The permission is then updated (if local), or placed on the correct IU-pair connection (if remote). **Step 5:** In the event that the base or end address of the shared region is not owned by the task, or the reconfiguration is interrupted, the IU goes into an error state, preventing all memory accesses until a reset signal is asserted.

Reconfiguration takes three cycles, with the local permission updated in the fourth cycle. Remote permissions take longer to become updated; a cycle is used to transfer the permission from one IU to another. The destination IU then takes a cycle to place this new permission into its new permission register before it becomes active. Further delays are incurred if the IU is already undergoing reconfiguration, or when a permission from a different IU-pair is being added causing a delay until the correct buffer is accessed.

The IU defaults to normal operation, checking all memory transactions. As soon as a task writes to the New Permission Register, the reconfiguration begins, and must be completed; if the procedure is interrupted, the IU goes in to the error state.

VI. Synthesis

Table 2: IU Synthesis (Total Logic Element Use)

1 task, 4 static permissions	237
2 tasks, 4 static permissions per task	262
3 tasks, 4 static permissions per task	275
4 tasks, 4 static permissions per task	295
2 tasks, 3 static permissions and 1 local shared permissions per task	562
3 tasks, 2 static permissions and 2 local shared permissions per task	700
2 tasks, 1 static permission, 1 local shared permission, 2 remote shared permissions per task	1361
1 task, 4 remote shared permissions	1280

We performed *Analysis & Synthesis* on several IU configurations, targeting the Altera Cyclone IV FPGA (EP4CE115F29C7). Altera Quartus II v15.1 was used. The total logic element use is presented in Table 2. We used the Avalon Memory Mapped interface as our test bus protocol.

As expected, the resource usage increases as we support more tasks, and more permissions; once we add support for local reconfiguration, the resource usage jumps, largely due to the control logic required, as well as additional comparators (to check that the IU is being accessed for reconfiguration). Remote management for permissions adds a significant cost again, largely due to the added FIFO (36% of the resource usage comes from the addition of this remote connection logic).

This indicates that our security additions should be tailored to our specific design; IUs can be optimized so that they only contain the capabilities and permissions for tasks that have a relationship with each other. Additionally, the granularity of protection regions can be reduced by checking only the upper bits of memory transactions reducing the complexity of permission checking and the size of permission lists. In this early prototype it is worth noting that embedded memory bits were not used – storing permissions in this way will reduce logic element use of the FPGA. Furthermore, permission checks can also be serialised to reduce the cost of comparing the incoming transaction and the stored permissions.

VII. Related Works

To our best knowledge this work is the first to consider a highly decentralised approach to security in a heterogeneous multiprocessor system.

In single processor systems, TrustLite[11] and TyTAN[12] implement variations of PCBAC [10], where MPUs are extended to directly associate executable code regions with memory access permissions. Comparatively, these approaches provide higher protection granularity compared to the frequently used ARM TrustZone[13], which offers hardware supported isolation for only two "worlds" and a reliance on software for managing access control in each world. These works do not consider the implications of a heterogeneous hardware platform with IP blocks that can perform memory accesses, nor dynamic permissions.

In the domain of Network-on-Chips, several works propose "firewalls" at the network interface [14-18] which filter transactions. However, these approaches rely on an OS or other central authority to manage rules at run-time, such as in [15] where a dedicated processor is used. The approach proposed in [16] only allows processors with a supervisor mode the ability to request a permission change, which is limits the type of processors we are able to employ.

With respect to handling violations once they are detected, [18] proposes the reconfiguration of hardware firewalls to facilitate a heightened security mode where memory accesses are severely restricted. This work requires a complete system reset for continued execution, whereas our approach allows isolated task groups to continue running.

An alternative approach to isolation utilises cryptography; in [19], tasks are isolated by encrypting tasks with different keys, and tagging memory such that a task can only work with data it has created itself, or taken from the insecure part of the system. As expected with cryptography-based solutions, there is a large performance overhead. Another isolation strategy involves generation of schedules for temporal isolation [20], where a central loader dispatches tasks depending on the generated security policy.

In a similar bid to remove reliance on "trusted" software, the work presented in [21] illustrates an architecture for cloud infrastructure, whereby virtual machines (VM) are loaded and managed by hardware components.

VIII. Conclusions

Heterogeneous MPSoCs provide a security challenge; with various types of processors and IP blocks, it is necessary to design security aspects into both the application and execution platform, especially when a single OS cannot easily be deployed across the entire system. In this work we presented a system-level security approach that distributes and delegates responsibility for managing access control permissions to tasks within an application. This security approach is suitable for tasks that share resources, particularly when it is not appropriate to simply classify tasks as secure or non-secure. We avoid reliance on a central authority for managing permissions at run-time, instead exploring architectural support for permission checking and granting/revoking in the form of Isolation Units. Our future work includes further tailoring of the IU design based on task requirements, and architectural exploration of interconnect architectures for permissions sharing between IUs.

References

[1] M. Paulitsch, O. M. Duarte, H. Karray, K. Mueller, D. Muench and J. Nowotsch, "Mixed-criticality embedded systems – A balance ensuring partitioning and performance," in *Euromicro Conf. On Digital System Design (DSD),* 2015.

[2] S. Lukacs, A. Lutas, D. H. Lutas and G. Sebestyen, "Hardware virtualization based security solution for embedded systems," in *IEEE Int. Conf. On Automation, Quality and Testing, Robotics,* 2014.

[3] (12 May 2013). *SE Linux Project*. Available: http://selinuxproject.org/

[4] D. Ferraiolo and R. Kuhn, "Role-based access controls," in *In 15th NIST-NCSC Nat. Computer Security Conf.,* 1992

[5] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE Communications Magazine,* vol. 32, 1994

[6] L. Szekeres, M. Payer, Lenx Tao Wei and R. Sekar, "Eternal War in Memory," *IEEE Security & Privacy,* 2014.

[7] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis and S. Ioannidis, "You can type, but you can't hide: A stealthy GPU-based keylogger," in *EuroSec'13,* Prague, Czech Republic, 2013.

[8] D. Kleidermacher and M. Kleidermacher, "Chapter 3 - secure embedded software development," in *Embedded Systems Security*, Oxford: Newnes, 2012

[9] A. Hattendorf, A. Raabe and A. Knoll, "Shared memory protection for spatial separation in multicore architectures," in *IEEE Int. Symp. on Industrial Embedded Systems (SIES'12),* 2012

[10] R. Strackx, F. Piessens and B. Preneel, "Efficient Isolation of Trusted Subsystems in Embedded Systems," in *Int. ICST Conf., SecureComm*, Singapore, 2010.

[11] P. Koeberl, S. Schulz, A. Sadeghi and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems,* Amsterdam, 2014

[12] F. Brasser, B. El Mahjoub, A. Sadeghi, C. Wachsmann and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Proceedings of the 52nd Annual Design Automation Conference,* San Francisco, 2015

[13] ARM Limited, "ARM security technology - building a secure system using TrustZone® technology," ARM, Tech. Rep. PRD29-GENC-009492CUnrestricted, 2009.

[14] J. Porquet, A. Greiner and C. Schwarz, "NoC-MPU: A secure architecture for flexible co-hosting on shared memory MPSoCs," in *Design, Automation & Test in Europe Conference (DATE),* 2011

[15] M. LeMay and C. A. Gunter, "Network-on-chip firewall: Countering defective and malicious system-on-chip hardware." in Logic, Rewriting, and Concurrency, 2014.

[16] L. Fiorin, G. Palermo, S. Lukovic, V. Catalano and C. Silvano, "Secure Memory Accesses on Networks-on-Chip," *IEEE Transactions On Computers,* vol. 57, 2008.

[17] M. D. Grammatikakis, K. Papadimitriou, P. Petrakis, et al., "Security in MPSoCs: A NoC Firewall and an Evaluation Framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 34, 2015.

[18] P. Cotret, G. Gogniat, J. P. Diguet and J. Crenne, "Lightweight reconfiguration security services for AXI-based MPSoCs," in *22nd Int. Conf. on Field Programmable Logic and Applications (FPL),* 2012.

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *SIGPLAN Not.,* vol. 35, 2000.

[20] L. A. D. Bathen and N. Dutt, "PoliMakE: A policy making engine for secure embedded software execution on chip-multiprocessors," in *Proc. of the 5th Workshop on Embedded Systems Security,* Scottsdale, 2010.

[21] E. Keller, J. Szefer, J. Rexford and R. B. Lee, "NoHype: Virtualized cloud infrastructure without the virtualization," in *Proceedings of the 37th Annual Int. Symp. on Computer Architecture,* Saint-Malo, 2010.