PoliMakE: A Policy Making Engine for Secure Embedded Software Execution on Chip-Multiprocessors

Luis Angel D. Bathen, Nikil Dutt
Center for Embedded Computer Systems
University of California, Irvine
{lbathen, dutt}@uci.edu

ABSTRACT
Secure software execution on chip-multiprocessor platforms is compromised by threats such as software-based side channel attacks that expose information from shared memory. The increasing amount of shared (memory or computational) resources on emerging chip-multiprocessors further exacerbates security threats, highlighting the need for secure policies to manage on-chip resources. We present PoliMakE, a methodology that enables exploration and generation of customized policies to guarantee secure software execution on a chip-multiprocessor system in the presence of software-based side channel attacks. PoliMakE analyzes an application's security needs and generates a series of custom policies that dictate how to safely execute tasks and efficiently manage the computational, communication, and memory resources. Our experimental results on DRM, JPEG as well as some synthetic applications show that PoliMakE enables secure software execution with minimal performance overhead, while reducing power consumption, since the policies are customized to efficiently utilize the available on-chip resources. For the case study of running DRM in secure mode concurrently with JPEG encoding, we are able to observe 61% performance improvement when compared to standard approaches. Our policy generation engine is able to generate policies in only a matter of minutes for secure applications with hundreds of tasks. Unsecure applications were observed to resume execution up to 99% faster than with the traditional halt approach.

Categories and Subject Descriptors
C.1.4 [Processor Architectures]: Parallel Architectures - Mobile processors; C.3 [Special-purpose and Application-based systems]: Real-time and embedded systems; D.3.4 [Programming Languages]: Processors - Compilers; C.4 [Performance of Systems]: Modeling techniques; Performance attributes; D.4.6 [Security and Protection] : Access Controls; Security Kernels

General Terms
Algorithms, Performance, Design, Security

Keywords
Data Reuse, Estimation, Multiprocessors, Security, Scheduling, Scratchpad Memory, Streaming Applications, Task Mapping, Policy, Isolation

1. INTRODUCTION
Embedded systems are often used to create, store, and manipulate sensitive data, whose security risk increases as new features continue to emerge in the mobile embedded system domain such as network connectivity, application downloading, migration to multiprocessors, etc. With network connectivity, embedded systems are exposed to similar risks (virus, worms) as those seen in the PC domain. Application store services -- which allow users to download and install applications -- further expose the embedded systems to installation of malicious software. Embedded Chip-MultiProcessors (CMPs) have emerged as a viable platform to overcome uniprocessor limitations [1], technology scaling challenges, and the demand for media-rich power and performance aware computing. CMP platforms have been shown to perform 50-100% better than superscalar architectures for applications with high levels of parallelism [2] and CMP platforms are well suited for emerging multimedia applications with high levels of parallelism such as JPEG [3], JPEG2000 [4], and H.264 [5], where data partitioning exposes significant task level parallelism [6]. CMPs often contain heterogeneous on-chip memory hierarchies composed of small caches and/or software-controlled scratch-pad-memories (SPMs), where SPMs are favored over caches due to their increased predictability, and reduced area and power consumption [7]. However, this increase in on-chip real-estate and the ability to share resources opens the door to new threat models (e.g., side channel attacks [9]) that were not present in the uniprocessor domain. Any one of these vulnerabilities may lead the embedded systems to (a) run a malicious application that tries to access private information via software attacks such as buffer overflows [8], or (b) expose private information via side channel attacks [9].

Much work in embedded systems security has focused on preventing software based attacks rather than physical level attacks (and some side-channel attacks) because software based attacks are the most significant threat, given that they can be achieved with less complexity, and there is no need to have “physical” access to the device. Therefore, for this work, we do not focus on preventing attacks which employ intrusive methods that rely on physical accesses.

There are several ways of protecting sensitive data in an embedded system: simple buffer overflow protection via stack guard [10], complex hardware monitors [11] which rely on co-processors to monitor the execution of software at the cost of k-extra co-processors, and resource isolation [12]. In this paper we present PoliMakE, a methodology that enables exploration and generation of customized policies to guarantee secure software
execution on a chip-multiprocessor system in the presence of software-based side channel attacks. PoliMakE analyzes an application's security needs and generates a series of custom policies that dictate how to safely execute tasks and efficiently manage the computational, communication, and memory resources. Designers may choose to select policies that will satisfy their power and performance needs. Policies are also enforced at run-time by our run-time secure scheduler. The run-time secure scheduler will determine which policy to load, given the CMP load, and the device settings (e.g., power or performance). At the core of our policy enforcing scheduler is the idea of on-chip application sandboxing (isolation) to guarantee secure software execution, with minimum performance overhead.

The remainder of this paper is organized as follows: Section 2 motivates our work. Section 3 gives an overview of PoliMakE. Section 4 presents software driven policy making, and Section 5 describes secure policy enforcement. Section 6 presents related work. Section 7 describes our experimental results, and Section 8 concludes the paper.

2. MOTIVATION

2.1 Software Attacks
Many software based attacks focus on exploiting vulnerabilities in legacy C library code such as strcpy(), where failure to check bounds causes buffer overruns, which can be used to overwrite memory addresses with carefully crafted payloads. These payloads insert return addresses to malicious code which are executed on return to the function. In 2004 alone, the Department of Homeland Security reported 323 buffer overflow vulnerabilities [13, 14]. These attacks require minimal effort in distribution as all that it takes, is a well crafted piece of malicious code to make it to a well known website or on a well known third party application store, from which users download/install the application (eventually running it). Simple exploits can be used to steal secrets from legitimate running applications as shown in [8], where exploiting the strcpy() function results in the attacker obtaining the key of a DRM [15] system.

2.2 Side Channel Attacks
Side channel attacks range from very sophisticated differential power analysis attacks (DPA) [16], where power analysis traces are obtained from probing a device at the voltage inputs/outputs and measuring the voltage thereby stealing the crypto algorithm’s keys, to software oriented timing analysis attacks [9]. The migration of embedded platforms from uniprocessor to on-chip multiprocessors opens new vulnerabilities and risks from side channel attacks due to sharing of on-chip resources in an attempt to improve system-wide performance. A side channel attack against an AES system was shown in [9], which allows the attacker to gather “leaked” information about the keys, thereby allowing the attacker to infer and steal the secret key. Such attack was made possible because of the fact that the memory resources were shared among different processing elements.

2.3 Need for isolated execution
Isolation has been shown to be effective in preventing attacks in many different domains. From language based isolation [17] (where code can be verified prior to execution), to sandboxing [18] (where tasks are isolated and kept contained within a domain), virtual machines (VMs) can enforce isolation by ensuring that each VM’s resources may be visible only to the host OS. OS-kernel [19] based isolation or trusted code bases (TCBs) use minimalistic chains of services for system execution. TCBs tend to be small so that they can be verified and used to sign/verify applications prior to their execution. Finally, hardware based isolation [12] has been used to isolate faults to a given core in a multiprocessor environment, where the potential of running malicious applications with applications that operate over sensitive data poses significant threats. Here, isolation techniques such as sandboxing are extremely beneficial, since task isolation ensures that neither can the malicious application access resources used by the trusted application, nor can a compromised trusted application access resources from outside of its own “box” [20].

2.4 Contributions
In this paper we propose PoliMakE, a methodology for secure software execution in the presence of both software based attacks (i.e. buffer overrun), and side channel attacks (i.e. cache timing attacks) via policy making and enforcing. To the best of our knowledge, PoliMakE is the first approach to guarantee secure software execution for on-chip multiprocessors through generation of customized policies. We also propose a means to enforce these policies via sandboxing, and dynamic policy selection. The target chip-multiprocessor (CMP) platform requires very minimal changes, a secure DMA, and arbiters that follow ACLs (Access Control Lists) when performing transactions on behalf of their masters. Our methodology for the analysis and generation of security requirements to enable secure execution of an application on CMPs has several contributions:

- We customize policies to guarantee secure software execution in a multi-processor environment. Our policies consist of a secure schedule, memory mapping, and set of requirements, which are to be enforced at run-time.
- We generate a run-time scheduler that enforces our policies via sandboxing in order to guarantee that execution of an application follows their policies.
- We propose a methodology to make and enforce secure policies given different embedded system requirements such as power or performance.
- Our security policies may be used along other methods to guarantee secure software execution, since the necessary changes to the architecture to enforce them is minimal.

3. POLIMAKE OVERVIEW
We first describe the target architectural platform for PoliMakE and then describe the PoliMakE framework in detail.

3.1 CMP Platform

![Figure 1. CMP Architecture](image)

Figure 1 shows our CMP architectural model that employs a
simple homogeneous CMP architecture, running low power RISC cores, each with access to a series of local Scratch Pad Memories (SPMs), instruction cache, and a DMA engine to facilitate the data transfers among the various SPMs and main memory [21]. We chose to support a shared bus communication infrastructure since it is the most commonly used in today’s embedded systems (i.e., AMBA AHB bus protocols). In order to guarantee secure execution of an application, we enhanced our CMP models with a secure DMA that can be programmed to transfer data from/to secure/unsecure memory regions by authorized users. We implement a simple ACL (Access Control List) model at the arbiters and DMA. The ACLs are tied to hardware CPU IDs, to prevent them from being spoofed. As described in the next section, the ACLs are generated by PoliMaKe on a per-buffer basis, tying an ACL to a given buffer and memory region, and loaded when the application’s policies are fetched by the secure run-time scheduler.

3.2 PoliMaKe Framework

Figure 2. PoliMaKe Framework

Figure 2 shows the overall PoliMaKe framework, composed of three main components:

- The front end, which transforms the application’s source code in order to maximize CMP utilization. Our front end is based in the work done in [21], which takes an application, extracts their kernels, finds inter-kernel reuse opportunities (to reduce unnecessary off-chip accesses), and generates an augmented task graph. The augmented task graph goes through node clustering (driven by data-reuse opportunities), where kernels that operate over the same data set are merged.
- The middle end, which analyzes the application’s source code, and determines its security requirements with respect to secure compute, memory, and communication resources.
- The back end (Policy Making Engine), which takes the application’s security requirements as input, and uses them to generate the application’s policies, which consists of a secure schedule, a data map, and the resources to be utilized when the application and its policy are loaded.

3.2.1 Policy Making Engine

The heart of PoliMaKe is the Policy Making Engine in Figure 2, which takes the application’s security requirements and generates customized policies for the given platform to meet different performance, and power requirements. Once policies are generated, they are packaged along with the application, and signed. When a user downloads the application, both the application executable as well as their policies are verified, and stored in encrypted format. Since our main focus is the generation and enforcement of the application’s policies, we will not go into details on how applications are purchased/downloaded, for now, let us assume that they are distributed by a trusted third party.

3.2.2 Basic Assumptions

We assume that the application has been initially partitioned into high level tasks, which are then analyzed by our policy making engine. We also assume that the designer can generate initial security requirements for a given application, for instance, determine which buffers/variables need to be protected (i.e., a device key for a DRM system). For the Run-time Secure Scheduler we assume a simple round-robin scheduler, which keeps a queue of ready tasks, and executes them one by one. When a call to a trusted application is made (e.g., playback of a DRM ring tone or music file), the DRM executable as well as their policies are loaded. Given a policy, the scheduler then decides what resources need to be allocated to DRM, and isolates its execution.

4. SOFTWARE DRIVEN POLICY MAKING

As shown in Figure 2, at the heart of PoliMaKe is the policy making engine. The policy making engine requires the application to be analyzed for security requirements prior to generating the policies.

4.1 Establishing Security Requirements

The first step in policy making is establishing security requirements for a given application. We assume the application developer will identify and define the set of variables/arrays that need to be secured (protect their confidentiality: i.e. keys). The static analyzer then determines the necessary security buffers, processing requirements, and secure communication channels required to support secure access to these data elements.

4.1.1 Initial security analysis

Figure 3 (a), shows two tasks, t1 and t2, where each task has a well defined kernel. The user has specified that the array a[i] needs to be secure, so it is part of the global security list (g_s_lst: a). Once we have parsed the list of variables to secure, we then proceed to generate security dependence graphs between secure buffers and variables/buffers affected. Our secure def-use chain analysis mirrors the approached proposed in [22]. As shown in Figure 3 (b), our analysis determines that variable d is affected since a is a secure element, so it is added to the local security list (l_s_lst: d). Figure 3 (c) shows the memory map of each of the buffers. This is useful when determining secure memory regions. Figure 3 (d) shows the initial task graph.
4.1.2 Security task graph augmentation

Figure 4. Augmented task graph with local buffers

The next step in our approach is augmenting the initial task graph with our security information. Figure 4 shows the initial task graph with an additional set of edges (dashed edges), which determine the memory requirements for each node. For each buffer, we generate an edge with the following information:

- Address of first access
- Address of last access
- Buffer size
- Access type
- Local or shared
- Whether it is a secure buffer or not

We assume that tasks can be statically analyzed for buffer requirements, so we can generate the initial memory mapping ahead of time. In cases where buffers are dynamically allocated, we omit regions (start/end addresses) from the security edges, and let the run-time scheduler take care of enforcing accesses to these buffers by using Access Control Lists (ACLs).

4.1.3 Determine shared buffer information

Figure 5. Add inter-kernel shared buffers

The next step is determining inter-kernel shared buffer requirements. This information is useful when security buffers are shared between kernels, and we want to minimize off-chip accesses. In the case where a is mapped to a given memory, when t1 ends its execution, it is shredded from the local memory, and if modified, offloaded to main memory. Since t2 reuses part of a, we split accesses to the buffer into local and shared accesses, which can be identified by the edges given by: a1 ∩ a2: <0, 63, 64, r/w, shared, sec> and a2: (a1 ∩ a2)=<64, 127, 64, r/o, local, sec>. Once sharing is determined, we then update the task graph.

4.1.4 Security requirements generation

Finally, the security requirements for the application are derived. These security requirements determine the amount of secure memory space needed, means of communication between secure and unsecure memory regions. Figure 6 shows the requirements graph between tasks and their buffers. Dotted lines mark secure channels, and dotted circles mark secure buffers, where as straight edges and circles mark unsecure buffers and channels. Table 1 shows a brief summary of the requirements in terms of secure buffers and channels for each task.

![Figure 6: Security Requirements](image)

### Table 1. Security Requirements

<table>
<thead>
<tr>
<th>Secure Buffers</th>
<th>Unsecure Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2: &lt;64, 127, 64, r/o, local, sec&gt;</td>
<td>b: 128 – Local (t1)</td>
</tr>
<tr>
<td>a1 ∩ a2: &lt;0, 63, 64, r/w, shared, sec&gt;</td>
<td>a1/ a2: 64 – Shared (t1, t2)</td>
</tr>
<tr>
<td>d – Local (t2)</td>
<td>d to t2</td>
</tr>
</tbody>
</table>

4.2 Secure Scheduling and Data Mapping

Once the security requirements have been derived from the application, the next step is to generate the full policy, which includes a secure schedule and memory allocation that must be
followed at run-time in order to guarantee the secure execution of the application. Our secure scheduler and data placement algorithm can generate policies customized to meet the designer’s objectives, e.g., performance, or power consumption. In cases where performance is the main goal, buffer reuse opportunities (regular and secure) may be ignored, and the scheduler will focus on generating performance-aware security policies. Similarly, when the designer chooses to generate power-aware security policies, the performance may be impacted as shared buffers may be reused by tasks, and secure/unsecure memory regions are deallocated only when the last task in the access list for the given buffer is done using it.

<table>
<thead>
<tr>
<th>Line #</th>
<th>Secure Scheduler Algorithm (Cont.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>update_ready_tasks_list(time)</td>
</tr>
<tr>
<td>45</td>
<td>{</td>
</tr>
<tr>
<td>46</td>
<td>for each task T in tasks</td>
</tr>
<tr>
<td>47</td>
<td>if all dependencies of T have been satisfied</td>
</tr>
<tr>
<td>48</td>
<td>add T to ready tasks list</td>
</tr>
<tr>
<td>49</td>
<td>update_platform_state(time)</td>
</tr>
<tr>
<td>50</td>
<td>}</td>
</tr>
<tr>
<td>51</td>
<td>update_platform_state(CPU, T, time)</td>
</tr>
<tr>
<td>52</td>
<td>{</td>
</tr>
<tr>
<td>53</td>
<td>for each CPU in platform</td>
</tr>
<tr>
<td>54</td>
<td>if CPU is busy</td>
</tr>
<tr>
<td>55</td>
<td>if current task T expected end time &lt;= time</td>
</tr>
<tr>
<td>56</td>
<td>unmap(T)</td>
</tr>
<tr>
<td>57</td>
<td>}</td>
</tr>
<tr>
<td>58</td>
<td>unmap(T)</td>
</tr>
<tr>
<td>59</td>
<td>{</td>
</tr>
<tr>
<td>60</td>
<td>for each buffer B in T</td>
</tr>
<tr>
<td>61</td>
<td>if buffer_reuse_enabled and B is secure</td>
</tr>
<tr>
<td>62</td>
<td>remove B</td>
</tr>
<tr>
<td>63</td>
<td>else // no reuse</td>
</tr>
<tr>
<td>64</td>
<td>remove B</td>
</tr>
<tr>
<td>65</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 8. Secure Scheduling and Data Mapping (Cont.)

The designer may also choose to define the maximum number of resources to utilize during the policy generation. Furthermore, the designer can explore different utilizations by generating a search space for tradeoff analysis, by assuming an unlimited number of resources. Thus as shown in Figure 2, PoliMakE supports the customized generation of multiple policies, with each policy tuned to be power-aware, performance-aware, meeting a complex cost function, or even generating a family of policies by varying the number of resources needed to execute the policy. The run-time scheduler will then use the right policy given the load of the CMP and the user’s requirements. For instance, if the user is running DRM enabled music files while the battery is low, a power-aware policy may be loaded.

Figure 7 shows the pseudo code for our security aware scheduler. Lines 1-20 go over the main algorithm which consists of a priority scheduler, that sorts ready tasks based on security requirements determined by resources needed (secure buffers, channels, etc.). The scheduler then decides whether or not it is worth to map the task at the current time (Line 12). The cost of mapping a task is given by the cost of loading the data at the current time for task $T$ on a given CPU with the current data mapping. We then compare this cost with the cost of not mapping task $T$ and choosing to map another task. Mapping a task (Lines 22-43) requires the scheduler to load buffers onto available SPM resources, program the DMA, and set up ACLs at the arbiters and DMA. In case there is no more SPM space, then tasks are mapped onto main memory. Secure buffers are mapped only onto secure SPMs. Prior to scheduling, the designer can choose to define the amount of real-estate to be secured, this includes the number of CPUs and SPMs.

Figure 8 shows the remainder of scheduler methods. The update to the ready tasks list (Lines 44-51) simply resolves dependencies for each task. The platform update, which refreshes the state of the system (Lines 53-60), checks each busy CPU, and determines which the task mapped onto it has finished its execution by the current time. If so, it marks the task as done, and unmaps it. The unmap method (Lines 62-70) determines whether to trash all buffers in the SPMs, or keep them, in case the reuse of secure buffers is enabled. Secure buffer reuse is accomplished by looking at the buffer’s ACL, and determining whether the current task is the last task to use it.

Power-aware security policies may require buffering sharing to be enabled, as off-chip accesses consume orders of magnitude more power than on-chip accesses [21, 23]. Power-aware policies may also affect performance, as the amount of real-estate is reduced by locking down buffers (memory regions). So if designers desire performance-aware security policies, they must switch off the secure buffer re-use flag.

4.3 SYN_SMALL Policy Example

Once the scheduler and data mapping algorithm has gone through the security-augmented task graph, we then proceed to generate the policies for the secure execution of the application. We illustrate this process (Figure 9) using a snapshot of a small secure application (called SYN_SMALL) to be executed consisting of five tasks ($t_0$ through $t_4$), and 8 buffers ($buf_5$ through $buf_{12}$). The dark arrows define execution dependence between tasks, the light (green) arrows show connectivity between tasks and insecure buffers (i.e., $t_2$ and $buf_5$ via $be_{2,5}$), and the red arrows connecting tasks to buffers (red boxes) show security critical buffers (i.e., $t_1$ and $buf_6$ via $be_{1,6}$). Note that connectivity between $t_1$ and $buf_6$ will require a secure channel of communication.
Figures 10 and 11 show snippets of the customized policies for the execution of application SYN_SMALL, tuned for power and performance respectively. The policies contain schedule information for each CPU, where each SCHED node contains start time, end time, and task mapping information. For each task, we then obtain buffer MAP information. For instance, Figure 10 shows the power-aware policy where task t2 has been mapped to cpu0, with start time of 0, and end time of 1643 cycles. Task t2 has a total of four buffers, three are secure buffers (buf_8, buf_9, buf_11) and buf_5 which is unsecure. Each buffer has MAP information which determines which memory it will be placed on (buf_5 is mapped to mem_2). MAP information for each buffer also contains the ACL, with different rights (read, write, or read and write). For instance, buf_5 can be read and written to by t2, but read only by t3. In the interest of brevity we have omitted extra information (i.e., address map) from these snapshots. Figure 11 shows a performance-aware policy for the same application (SYN_SMALL); note the extra MAP requests for t3, where buf_5, buf_8 and buf_12 are not re-used, thereby needing to be loaded back into the CMP. Sharing buffers locks memory regions for \( x \) amount of time, if \( x \) is large enough, it might prevent other tasks from utilizing the on-chip memory, hence, fetching data from off-chip memory might yield better performance.

### 5. Secure Policy Enforcement

Once PoliMakeE has generated a series of policies to meet the designer’s and application’s security, performance and power requirements, the next step is enforcing these policies.
and sandboxing, we efficiently isolate resources, and allow for access resources within the sandbox. By allowing policy driven application running outside of the sandbox would be unable to access any policy loaded. The application in lockdown would not be able to sandboxing would allow an application to go into lockdown execution of a

Figure 15 shows the effect of combining policies with on-chip sandboxing. As we can see in the example, the secure run-time scheduler decides to allocate 50% of the on-chip real-estate to the trusted application evicting the two tasks with less context switching overhead. Unlike the halt approach, where all other tasks need to wait for the trusted application to run, PoliMakE allows for both trusted and un-trusted applications to run concurrently, provided that the trusted application enters the lockdown mode. In this example, we see that the only penalty incurred for executing the trusted application and migrating tasks T1 and T2 is their context switch. Not only is this approach more efficient in terms of performance, but it also saves power, as the data for the other two tasks T3 and T4 did not have to be evicted from memory (thereby incurring no context switch penalties). So the average delay for this mode decreased from 460ms in the halt approach to 90ms in the on-chip sandboxing mode.

5.3 On-chip Sandboxing

As discussed in Section 5.2, a halt approach is not efficient in terms of performance and power. A more efficient approach generated by our PoliMakE framework would enable on-chip sandboxing, allowing both trusted and un-trusted applications to run concurrently, where trusted tasks are free to share data among themselves, without being at risk of some malicious application interfering with their communication or tampering with their data.

Figure 15 shows the effect of on-chip sandboxing on the execution of a trusted application. Starting with a CMP, sandboxing would allow an application to go into lockdown mode, where it is given a set of dedicated resources based on the policy loaded. The application in lockdown would not be able to access any resource outside of its sandbox, similarly, any application running outside of the sandbox would be unable to access resources within the sandbox. By allowing policy driven sandboxing, we efficiently isolate resources, and allow for trusted and un-trusted applications to co-exist within the same chip.

5.4 PoliMakE Driven On-chip Sandboxing

Figure 16 shows the effect of combining policies with on-chip sandboxing. As we can see in the example, the secure run-time scheduler decides to allocate 50% of the on-chip real-estate to the trusted application evicting the two tasks with less context switching overhead. Unlike the halt approach, where all other tasks need to wait for the trusted application to run, PoliMakE allows for both trusted and un-trusted applications to run concurrently, provided that the trusted application enters the lockdown mode. In this example, we see that the only penalty incurred for executing the trusted application and migrating tasks T1 and T2 is their context switch. Not only is this approach more efficient in terms of performance, but it also saves power, as the data for the other two tasks T3 and T4 did not have to be evicted from memory (thereby incurring no context switch penalties). So the average delay for this mode decreased from 460ms in the halt approach to 90ms in the on-chip sandboxing mode.

6. RELATED WORK

Secure software execution has been addressed in both the software domain, as well as the hardware domain. In this section we mention techniques that are most relevant to our work. There has been a large amount of effort in defending against software vulnerabilities such as such as CCured [24] which does bound checking on C code, StackGuard [10] for buffer overflows, Smashguard [25] to protect the return address of functions, Pointguard [26] to protect pointers from the buffer overflow vulnerabilities. Patel et al. [27] proposed SHIELD, a methodology to detect code injection attacks via hardware monitors. The extra CPU monitor looks at the control flow graph of the applications running, and tries to determine if a given basic block has deviated from its bounded execution time. In case a significant deviation occurs, a flag is raised. Zambreno et al. [28] proposed a methodology to protect software execution via an ad-on FPGA co-processor that is used to check execution by decrypting hidden keys within the regular unencrypted instructions. Shao et al. [29] proposed HSDefender, a hardware/software technique to protect embedded systems against buffer overflow attacks. Arora et al. [30] proposed an architectural support to transparently execute CCured checks. Coburn et al. [8] propose a security enhanced communication architecture that allows designers to use information in the communication infrastructure to enforce security protocols. Suh et al. [31] propose a processor platform that allows for the tamper-aware secure execution of applications within the processor. Lie et al. [32] propose a platform for isolated execution of software via execute-only memory, which prevents tasks from being modified. The ARM TrustZone [33] provides designers with the means to develop applications and execute
them in either secure or unsecure environments. Shimizu et al. [34] proposed the use of the cell processor as a secure vault, where each PE can potentially execute a task in isolation, with no access to the outside world. Agarwal et al. [12] proposed isolation in multiprocessor environments as a means to isolate software faults. Saputra et al. [22] proposed a methodology for analyzing secure data dependencies, and selectively executing each piece of data in either safe or unsafe mode. Most hardware monitoring approaches rely on profiling the applications, and gathering execution information which they then try to monitor at run-time. These approaches suffer from overheads in terms of area and performance. Area overhead is incurred by adding extra logic and hardware (including co-processors) to monitor processes, and performance overhead comes from the need to communicate with the monitor. Full platform support for secure software execution such as AEGIS, SECA, XOM, and ARM’s TrustZone is beneficial in case applications need to run securely, but if an application has no need for security support, then all the extra resources meant for guarantee security are wasted. The cell vault approach is limited in the sense that a single PE executes a task, but sharing between “vaults”, is not possible, which may degrade performance when you have tasks that need to run in secure mode, while sharing information. Current isolation approaches such as [12] do not go over how to isolate resources, and guarantee that applications are executed safely. To the best of our knowledge, we are the first to propose the idea of customized policy making to guarantee secure software execution for CMPs. We also propose a means to enforcing these policies via on-chip sandboxing, and dynamic policy selection. Our proposed platform requires very minimal changes, a secure DMA, and arbiters that follow ACLs when performing transactions on behalf of their masters.

7. EXPERIMENTAL RESULTS

7.1 Experimental Set-up
Polimake is built on top of our SystemC [35] simulation engine [23]. Our environment runs on Linux, kernel 2.6. The run-time of Polimake’s was obtained through profiling it on an Intel Core 2 Duo running at 2.0 GHz. We chose to evaluate Polimake’s efficiency by simulating the launch of a DRM application during the execution of the encoding of an image. We selected JPEG from the MediaBench II [36] suite. We based our DRM design according to DRM 2 standard [15]. The music file used was encoded with lameMp3 as 44.1 kHz 128 kbps j-stereo MPEG-1 Layer III (11x) qval=3. We also generated synthetic benchmarks to simulate small and large trusted applications. The image being encoded was Lena at 1024×1024, 24bit depth.

7.2 Effects of Sharing Secure Buffers
Figure 17 shows the effect of sharing and no-sharing on multiple CMP configurations for a trusted application running 37 tasks. Each data point refers to a CMP configuration. As we can see from the top graph, execution time is reduced as we increase the number of cores as well as the memory size. Similarly, off-chip accesses are reduced as we increase the number of resources. In general, secure buffer sharing reduces off-chip accesses which affect overall system power consumption. There are cases when sharing will degrade performance as is in the case of the 16CPU/4KB example, where no sharing did better. So if a designer wishes to consider performance as his/her main concern, he/she may choose to disable buffer reuse.

7.3 DRM Policy Different Utilizations
Figure 18 shows the normalized expected execution time for the DRM application given by policies with 25/50 % of dedicated resources, for the sharing and non-sharing cases. As we can see, for different configurations, we obtain different policies, which may lead to different performance numbers. We see great improvements when going from a single core to dual and minimal improvements when going up to five cores. At six cores we start seeing some performance degradation with respect to the 2-5 core cases. This is because the schedules have so many resources, that the policy ends up trying to utilize too many resources, which in turn rather than overlapping DMA transfers with execution of tasks, end up trying to launch multiple DMA requests in parallel and tasks are stuck waiting for access to DMA.

7.4 Effects of Polimake and On-chip Sandboxig
For this experiment we decided to evaluate the performance degradation for the JPEG encoder when a DRM instance is
launched at one million cycles into the simulation. We compare our approach to the halt approach, which grants DRM exclusive access to the underlying hardware. Figure 19 shows the partition of resources among JPEG and DRM. The first case is a dual core CMP, where JPEG is given a single core, and DRM goes into lockdown using the other core (JPEG_1 and DRM_1). Each bar represents a scenario, (No sharing halt, Sharing halt, No sharing PoliMaKe and sharing PoliMaKe). As we can see from this experiment, for the DRM, the execution time reduction incurred by PoliMaKe is about 7%, where as the regular music file (DRM M), shows up to 25% reduction. So we are able to show that PoliMaKe is most effective when the impact of the security application on the execution of other applications is greater.

### 7.7 PoliMaKe Run-time

Figure 22 shows the effect on run time as the number of tasks to schedule increase. So despite of the fact that the number of tasks went from 12 to 179 (15x increment), our runtime went up by less than 2x. Each run of PoliMaKe was in the order of a couple of minutes.

### 8. Summary and Conclusions

In this paper we proposed PoliMaKe, a methodology that explores, generates and exploits customized policy making to guarantee secure software execution in a multiprocessor system in the presence of software-based side channel attacks. To the best of our knowledge, we are the first to propose the idea of policy making to guarantee secure software execution for CMPs. We also propose a means to enforcing these policies via on-chip sandboxing, and dynamic policy selections. Our proposed platform requires very minimal changes, a secure DMA, and arbiters that follow ACLs when performing transactions on behalf of their masters. On our DRM/JPEG case study we observed up to 99% execution cycle reduction over the traditional halt approach. PoliMaKe is able to execute software securely in a transparent way with minimal execution overhead, with as little as 80 cycles due to smart context switching. Even in the extreme case of halting execution, following our secure schedule and memory mapping will improve performance and reduce power consumption, as policies are custom made to efficiently utilize the available resources. Future work includes integrating our scheduler into a virtualization layer that will manage a multiprocessor platform and enforce our policies.

### 9. ACKNOWLEDGEMENTS

This material is based upon work partially supported by the National Science Foundation Grants No. 0723955, CCF-0702797

### 10. REFERENCES


