2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines

Inheriting Software Security Policies Within Hardware IP Components

Festus Hategekimana, Joel Mandebi Mbongue, Md Jubaer Hossain Pantho, Christophe Bobda
CSCE Department
University of Arkansas
Fayetteville, Arkansas, USA
{fhategek,jmmandeb,mpantho,cbobda}@uark.edu

***Abstract*—Domain isolation enforcement is one of the challenging issues in software environments. To address this problem, NSA, in conjunction with the Secure Computing Corporation and the University of Utah, developed the open-source Flux Advanced Security Kernel (Flask), the mandatory access control (MAC) security architecture underlying major Operating Systems/Hypervisors widely deployed in cloud/desktop environments. In this work, we extend this security architecture to FPGA-based heterogeneous systems. Specifically, we explore the design and implementation of a security framework for controlled sharing of FPGA hardware modules in MAC-based OS/Hypervisor environments. The proposed design guarantees that hardware modules execute in the same security context as of the processes calling them by propagating the latter security policies expressed at the software level, down to the hardware. We prototype the proposed framework with SELinux and demonstrate its utility by evaluating trade-offs between security performance and execution overhead incurred by example applications. The preliminary results show our proposed framework provides isolation with an average of 0.6% worst case performance overhead.**

***Keywords*-Field Programmable Gate Arrays (FPGAs), Secure heterogeneous systems, Secure Execution.**

I. Introduction

Many of today's critical embedded systems are increasingly relying on FPGA-based system-on-chip (FPGA SoCs) because of the useful balance between the performance, scale, flexibility, and rapid time to market they provide [1]. This was recently exemplified with the recent Audi announcement that its 2018 A8 world's first Level 3 autonomous driving system will feature Altera's Cyclone FPGA SoCs for object and map fusion processing tasks [2]. Though, it is not just in embedded systems space where we are seeing advanced adoption of FPGA platforms. They are also being continuously integrated into cloud computing systems and data centers as evidenced by Amazon [3], Huawei [4], and Microsoft [5] recent announcements of providing cloud computing instances with FPGAs.

Despite this increased adoption, isolating mutually distrusted hardware modules in heterogeneous systems continues to be a strong area of concern. In this paper, we address the problem of design of a transparent security framework that propagates privilege boundaries enforced in software down, to individual hardware modules. Consider the classical FPGA utilization model (i.e. heterogeneous designs where software applications offload some of its computation-intensive tasks to the FPGA). One of the main security concern in such designs has often been that there is not any mechanism, inherent to the system and transparent to the application programmers, that ensures that data execution of one heterogeneous application's processes and that of their "callee" hardware modules are securely isolated from the others. If controlled sharing is not enforced, shared accelerators could act as a potential covert channels between caller processeses which reside in different security contexts (privileged vs. normal for instance). Current trends in FPGA adoption indicate that this problem will likely worsen as more systems developers turn to 3rd party developed "plug-and-play" IPs, IPs oftentimes whose security properties cannot be properly verified at time of use, to shorten development cycles. As a result, these trends can potentially lead to insecure hardware platform whose vulnerabilities can be exploited through software applications running on top [6], [7].

To address the problem of domain isolation enforcement in software environments, NSA, in conjunction with the Secure Computing Corporation and University of Utah, developed the open-source Flux Advanced Security Kernel (Flask) whose main objective is to provide flexible support for mandatory access control (MAC) policies in operating systems [8], [9], [10]. Today, Flask forms the foundation of the security architectures of major operating systems/hypervisor widely deployed in cloud/mobile/desktop environments (Linux, Android, Xen etc) [10], [11]. Our goal is to extend this security architecture to heterogeneous applications, in particular those incorporating FPGAs.

In this work, we borrow from the insights gained from domain separation and isolation of processes/VMs in MAC-based OSes/Hypervisors and apply them to ensure hardware modules isolation. Specifically, we explore the design and implementation of a transparent security framework for controlled sharing of hardware modules in CPU+FPGA heterogeneous systems developed on top of MAC-based OSes/Hypervisors (Linux and Xen in particular). The proposed framework goal is to guarantee that in such systems, hardware modules execute and reside in the same security context as the "caller" processes by propagating to the "callee" modules, caller processes' security privilege boundaries defined at the software level. Our design creates a generic security interface which allows our framework to be compatible with any Flask-enabled OS/Hypervisor. We implemented a prototype of our proposed framework architecture in x86-based Ubuntu SELinux (a Flask-enabled Linux) with attached Xilinx FPGAs where it manages all inter-processes HW modules communication according to Flask security policies. We demonstrate the utility of the proposed framework by evaluating HW modules execution overhead introduced by our framework implementation prototype. The results shows our proposed framework provides isolation with an average of 0.6% worst case performance overhead.

II. Background

We used Flask as the MAC-based security architecture throughout this paper. We chose Flask because it is a foundation to security kernels of the most widely deployed OSes/hypervisors, such as Linux and Xen.

A. *FLASK Security Architecture*

Flask security architecture was developed by the NSA, in conjunction with the Secure Computing Corporation and the University of Utah, with the main goal of providing flexible support for mandatory access control policies to operating systems [8], [9]. In system with MACs, a security label describing the security context is assigned to each system component, subjects (e.g. processes) and objects (files, sockets, memory segments, etc.); and all accesses between subjects and objects must be authorized by the policy based on the labels [8], [11]. Flask separates the definition of the security policy logic from the enforcement mechanism. The security policy logic is designed and implemented in specialized language such as XML by the system administrator and is compiled into binary loadable to the separate component of the OS/Hypervisor kernel, known as "security server", with interfaces for obtaining security policy decisions. Security policies enforcement is managed by "object managers" (e.g. process management, filesystem, sock IPC, etc). To minimize the overhead associated with permissions lookup, Flask allows object managers to maintain an access vector cache (AVC) component to store access decision computations which are provided by the security server for their subsequent usage.

III. Proposed Approach

In this section, we start by discussing the mechanism our framework uses to ensure controlled sharing of hardware modules. We follow this with a discussion on design considerations that guided our implementation choices.

A. *Hardware Isolation Model*

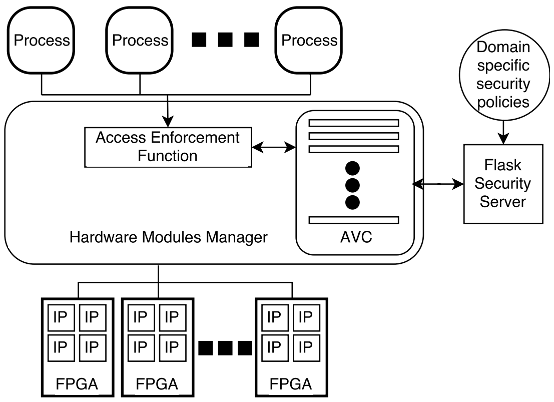

Figure 1. Hardware modules isolation framework

Figure 1 describes the architecture of our proposed hardware modules isolation framework. Our work considers heterogeneous systems usage model where processes executing at the software level offload their computation-intensive tasks onto hardware modules executing on a set of FPGAs. Our proposed isolation mechanism is enforced through the "Hardware Modules Manager (HMM)", an object manager for hardware modules objects. To enforce isolation, the HMM first acquires initial security contexts labels (SIDs) for N hardware modules it abstracts. It does this by sending access control information of the "caller" processes (i.e. SID for caller process, SID of caller process directory, and the class HW modules belong to) as parameters to the security server. The latter consults its security policies, determines security context for the callee HW module, and returns corresponding SID to the hardware modules objects manager. It then uses the returned SID to obtain access decisions by querying the security server with the caller process - callee HW SIDs pair and the requested Read/Write permissions. Flask security server consults its security policies and returns access decision to be enforced by the HMM. Access is granted only if the caller process and the callee hardware modules SIDs reside in the same security context and if that context allows permissions which are being requested. This makes it possible that processes which belong to a higher privileged level (i.e. higher security context) will not have to share the same hardware modules with processes which belong to a different security context, thereby ensuring proper hardware modules isolation.

B. *HMM Design*

We designed the HW modules object manager as software-hardware codesign. We partitioned the HMM access enforcement functionality along with the ability to remember access decisions in hardware and then partitioned in software, the interfaces for HW modules SIDs requests and policies update requests.

1) HMM Access Enforcement Function: The HMM access enforcement function is a simple hardware circuit which guards access to the hardware modules. It achieves this by adhering to the following procedure: When the HMM receives access request for the first time, the access enforcement function first requests the creation of the security context label of the callee hardware. This request is carried out by its software interface which calls Flask security server as discussed in the previous section. The access enforcement function then binds the returned SID to the callee hardware ID and stores this relationship in the HMM Access Vector Cache component. The latter will then be consulted by the enforcement function for subsequent requests to the same hardware ID, for deny/grant decisions and proceed to enforce them. During hardware execution, the HMM access enforcement function maintains in a specialized data structure, the list of busy hardware modules and their corresponding caller processes SIDs. This allows the access enforcement function to discard results of ongoing execution in case there have been changes in security policies mid execution, which the ongoing execution violates.

2) HMM Access Vector Cache: In order to minimize the overhead associated with access decisions requests and computations, during the initial access decision request, for the pair of SIDs provided, Flask security server provides more decisions than requested. The latter and the SIDs pair they map to, are then cached by the HMM's "Access Vector Cache (AVC)", the HMM component in charge of remembering access decisions. For this ability to scale without degrading the overall system performance due to expected repeated access requests, we partitioned this AVC capability into hardware. We then partitioned into the software, the AVC interface responsible for managing access decision misses and changes in security policies.

To manage access decision misses, the AVC queries its software component to request access decisions to the Flask security server. When the latter returns the decision, the AVC informs the HMM access enforcement function and updates its access decisions table. When there are changes in security policies, Flask security server alerts the AVC software component of policy changes. The latter notifies the AVC hardware component and proceeds to update its permissions state. Then, the AVC alerts HMM access enforcement function of changes in policies. The HMM access enforcement function proceeds to reevaluate the security context of ongoing hardware execution and discards the results of the execution if, per the updated policies, the execution was not authorized. The AVC software component informs the security server when policies update propagation is successfully completed.

C. Design Considerations

Communication costs between the caller process and the callee hardware module constitutes one of the major bottlenecks for FPGA-based heterogeneous systems. Adding privilege boundaries check operations on top of existing communication costs can be estimated to exacerbate these concerns. To mitigate this, we use "speculative execution". We speculate that the request may pass the security context boundaries check and we proceed to temporarily allow access to the hardware module while security context check goes on in parallel. Results from this execution are held until it is confirmed that the caller process is indeed authorized to access the hardware module. In case the caller process is found not authorized for this access, the results of the execution are nullified and an error handling routine is invoked. All failed access requests are logged, parsed, and reported to avoid denial of service scenarios where components continuously attempt illegal accesses.

IV. Implementation And Evaluation

In this section, first, we go through the user application compilation flow. Second, we discuss a prototype implementation of the isolation framework. Finally, we evaluate the implementation performance in context of real world heterogeneous applications.

A. Prototype Implementation

1) User Application Compilation Flow: Figure 2 illustrates the compilation flow of the user application. We assume the user is not aware of the presence of accelerators, but there are in their application code, computation-intensive tasks F(x) which can be offloaded onto accelerators. Our compilation tool takes the user application code, parses it, looks for keywords indicating the functions to be offloaded to accelerators, and replaces these functions with calls to the hardware functions, H(x), available to the FPGAs. To avoid CPU busy waits, the compilation tool adds an interrupt service routine which will be asserted when offloaded computations are done.

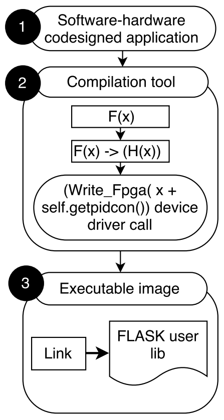
1
Software-hardware codesigned application
2
Compilation tool
F(x)
F(x) -> (H(x))
(Write_Fpga(x + self.getpidcon()) device driver call
3
Executable image
Link
FLASK user lib

Figure 2. User application compilation flow

In addition to these changes, the compilation tool also adds code for obtaining security context information of the caller thread. This information is then added to the original data that the user is sending to the accelerator. This newly created data package constitutes the data which will be sent to the driver. The compilation tool then builds the new user application code together with Flask user library to generate the binary ready to run on any Flask-enabled heterogeneous system.

2) Hardware Modules Manager: As we discussed in previous section, we designed and implemented the HMM Access Enforcement Function part hardware part software. The HMM generates SID labels for HW modules as they are requested. The first time the HMM receives access request to a HW module, it temporarily allows the request to proceed and initiatesin parallel, a setup process in which the HMM Access Enforcement Function software component generates a security context label for callee HW module. Once the SID and associated permissions are returned, the HMM Access Enforcement Function binds this information to the **[CallerProcess-SID | CalleeHardwareID]** pair and caches this data to the Access Vector Cache component. While the execution goes on, the HMM Access Enforcement Function hardware component queries the AVC to confirm the legality of the ongoing execution. The results of the execution are returned to the caller process only when the security policy authorizes the transaction. Otherwise, error code is returned. For subsequent requests, the HMM Access Enforcement queries the AVC with the access descriptor **[CallerProcessSID | CalleeHardwareID | Access Op]**. The AVC walks its content addressable memory (CAM) and looks for matching entry and returns the access ruling, a single **[Grant/Deny]** bit. In case of a miss, the AVC interrupts its software component with a fetch request to the Flask security server. The latter consults its security policy and returns the access decision to the AVC, which in turn updates its CAM and forwards the access decision to the Access Enforcement Function.

3) Security Policy: In this example prototype, we did not implement any special security policy outside of the default Type Enforcement and Multi Level Security (MLS) policies Flask provides. The latter defines list of user domains, object types and categories, and sensitivity levels. In this example, we concerned ourselves with only categories enforcement. For example, `[system_u:system_r:kernel_t:s0:c3.]` meant that processes which belong to `system_u` domain can only access HW modules marked with category c3, trusted HW modules category. You can further fine-tune security levels and create tighter objects categories, but these would be security policies management tasks, which is beyond the scope of this work..

Table I
HMM's Security Boundary Check Overhead

Calls	Exec. Time (FPGA+CPU+COM)
SID label creation	0.008μs + 35μs + 86μs
Decision Lookup (AVC Hit)	0.02μs
Decision Lookup (AVC Miss)	0.02μs + 35μs + 86μs
Decision Administration	0.012μs

B. Evaluation

1) Evaluation Setup: The evaluation was conducted on a host machine with Intel Core i7-5930K processor clocked at 3.60GHz. The host CPU runs Ubuntu 14.04.02 LTS with Linux Kernel 4.8.0-41 version. A PCIe Gen3 Upstream card with 4 Kintex UltraScale XCKU060 attached FPGAs is used. Each FPGA operates at 250MHz and contains 4 accelerator functions that the user application can call.

2) Execution Times Analysis: Table I shows average times spent on computations (CPU, FPGA) and communication costs (CPU-Driver-FPGA Mem.) during security boundary check by HMM, independent of the application. As the table indicates, first time accesses suffer (worst case), but the effect is quickly mitigated by caching labels and associated access rules for future usage. Since HMM allows speculative execution while security checks go on in parallel, applications will only suffer on balance, from the access decision administration overhead (0.012us or 3 FPGA Clk Cycles). Table II illustrates worst case execution times in context of real world heterogeneous applications. As the table indicates, the more there is parallelism to offload on HW modules, the less significant security check overhead becomes.

Table II
TOTAL EXECUTION TIMES (CPU+FPGA+COM) WITH AND WITHOUT ISOLATION ENFORCEMENT (UNDER FIRST-TIME ACCESS ONLY)

Benchmarks	**Inputs Size (Bytes)**	**Total Time W/out Iso.**	**Total Time W/ Iso.**	**Time Spent on FPGA**
Matrix Multiplication	64x64	38043μs	38113μs (+0.18%)	30.5μs
Inner Product	256x1	11097.5μs	11207μs (+0.99%)	5.5μs
Convolution 2D	64x64	20737μs	20743μs (+0.03%)	148μs
Vector Addition	256x1	11147μs	11268.01μs (+1.08%)	5μs

3) Code Complexity Analysis: Adding hardware calls and Flask library calls grew the user application code footprint by about 3Kb and incurred extra 0.16s of compilation time. Our software interface for the Hardware Modules Manager adds 250 lines of code to the Linux Kernel. The interface is generic and is compatible with any Flask MAC-enabled OS/Hypervisor. Although further optimization should be pursued to reduce the code footprint, we assess these one time costs are negligible since they do not affect runtime behavior of the applications.

V. RELATED WORK

While there have been attempts at providing isolated execution of hardware modules before, [12], [7], [13], none of them have targeted our assumed FPGAs usage model (i.e. multiple tenants sharing sets of FPGAs as part of heterogeneous applications). Most of the work in hardware isolation have generally focused on the problem of isolating IP cores execution in standalone applications running on FPGAs. There are a few works which discuss FPGA integration in the cloud that deal with the isolation issue [1], [14]. These works limit the discussion only on isolating accelerators to accessing virtual memory space of virtual machine which owns them. At the time of this writing, we are not aware of any works where hardware modules inherit real-time security context boundaries of processes calling them.

VI. CONCLUSION

This work presented the design and implementation of a security framework for controlled sharing of hardware modules in FPGA-based heterogeneous systems developed on top of MAC-based OS/Hypervisors. Preliminary results showed that the proposed security framework achieves isolation with an average of 0.6% worst case performance overhead.

REFERENCES

[1] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling fpgas in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. New York, NY, USA: ACM, 2014, pp. 3:1–3:10. [Online]. Available: http://doi.acm.org/10.1145/2597917.2597929

[2] Intel, "Report: Intel inside new audi autonomous car system," 2017.

[3] Amazon, "Amazon ec2 f1 instances," 2017.

[4] Xilinx, "Xilinx powers huawei fpga accelerated cloud server," 2017.

[5] Wired, "Microsoft bets its future on a reprogrammable computer chip," 2016.

[6] DARPA, "System security integrated through hardware and firmware (ssith)," 2017.

[7] F. Hategekimana, T. Whitaker, M. J. H. Pantho, and C. Bobda, "Shielding non-trusted ips in socs," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–4.

[8] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The flask security architecture: System support for diverse security policies," in *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, ser. SSYM'99. Berkeley, CA, USA: USENIX Association, 1999, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251421.1251432

[9] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 29–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=647054.715771

[10] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn, "Building a mac-based security architecture for the xen open-source hypervisor," in *21st Annual Computer Security Applications Conference (ACSAC'05)*, Dec 2005, pp. 10 pp.–285.

[11] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android."

[12] T. Huffmire, B. Brotherton, N. Callegari, J. Valamehr, J. White, R. Kastner, and T. Sherwood, "Designing secure systems on reconfigurable hardware," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 3, pp. 44:1–44:24, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1367045.1367053

[13] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine, "Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, May 2007, pp. 281–295.

[14] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "Fpgas in the cloud: Booting virtualized hardware accelerators with openstack," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 109–116.