CASTLE: Architecting Assured System-on-Chip Firmware Integrity

Sandip Ray, Atul Prasad Deb Nath, Kshitij Raj, and Swarup Bhunia
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611. USA
sandip@ece.ufl.edu, atulprasad@ufl.edu, kshitijraj@ufl.edu, swarup@ece.ufl.edu

***Abstract*—Modern System-on-Chip (SoC) designs include a large number of embedded microcontrollers that execute custom firmware. Firmware provides the flexibility of updating security features, *i.e.*, it enables *patching* or *in-field update*, in response to an emerging security threat, bug, or changing requirements. Unfortunately, current firmware update mechanisms are complex, manual, and error-prone. In this paper we present CASTLE, an architectural framework to enable systematic and assured updates to SoC firmware. The main workhorse of CASTLE is a centralized, dedicated IP in the SoC that is responsible for receiving, authenticating, and installing a patch. The architecture works with off-chip firmware validation flows, *e.g.*, cloud-based service for validating a proposed patch, and identifying compatibility constraints on other resident firmware in the SoC. The result is a comprehensive infrastructure that works seamlessly across architectures, vendors, and service providers, while meeting deployment and usability requirements. We demonstrate the application of proposed framework in addressing functional and security flaws of existing firmware patching mechanisms including firmware incompatibility, inadequate authentication, and time-of-check vs. time-of-use (TOCTOU) constraints.**

***Index Terms*—Security architecture, Firmware patching, Security policies**

I. Introduction

Modern SoC designs include many hardware IPs that implement a significant portion of system functionality through firmware. Firmware implementations have several advantages compared to traditional custom hardware, including the ability to update (1) during post-silicon debug in response to bugs (in either the firmware itself or other hardware components) without going through expensive re-spins, and (2) in field, in response to errors discovered after deployment or changing requirements. As we move towards devices with long field-life (*e.g.*, automotive systems, IoT, critical infrastructure components, etc.), the need for post-silicon, in-field update is getting increasingly critical. Consequently, there has been a significant upsurge in the number and diversity of microcontrollers in SoC designs, as well as the complexity of the firmware. In a commercial SoC design, it is typical to find at least a dozen embedded processors, often with heterogeneous ISAs running programmable firmware.

On the other hand, the firmware itself can be notoriously error-prone. Unlike conventional software, firmware is developed concurrently with hardware and must be functional before the hardware shipment. Consequently, a stable silicon (or even emulation/FPGA) platform is not available until late in firmware development. Firmware validation uses various virtual platform (VP), i.e., software abstractions of the underlying hardware platform, to cope with the unavailability of stable silicon platform of the target SoC. However, since these abstractions simplify or remove many details of the hardware functionality, validation with them misses a number of corner-case bugs. Furthermore, the correctness of a firmware code executing on a specific microcontrolled IP depends on correctness of other IP blocks (often running their own custom firmware) with which it communicates.

In spite of its criticality, we are aware of no disciplined approach for ensuring secure execution of firmware-controlled SoC designs. There has been some work on firmware analysis through dynamic (simulation-based) analysis and formal methods (See Sectin VI). These approaches do not account for various hardware corner cases or communication of the firmware with other IPs and suffer from scalability issues, and can typically only validate shallow properties under highly constrained environment. In practice, runtime monitors are introduced to ensure system-level properties, *e.g.*, it is common to introduce monitors to do runtime check for violation of privilege level escalation, unexpected access request for a sensitive memory address space, etc. However, such runtime monitors are also developed in an ad hoc manner, based on designer and validator experiences.

In this paper, we take the position that it is possible to develop high-assurance SoC designs even within the ecosystem of extensive and potentially untrusted firmware by providing architectural support to ensure runtime resilience of the system against compromises due to firmware bugs. The key idea is that such resliency requirements can be formulated as SoC security policies [1] and addressed through systematic policy enforcement paradigms. We demonstrate the viability of this vision through a specific architectural framework for ensuring firmware integrity in the context of in-field updates. Our framework CASTLE (**C**entralized **A**rchitecture for **S**ystema**t**ic Firmware **L**oad and Firmwar**e** Patch) is an extension of a centralized policy implementation architecture we developed in previous work [2]–[5]. The security architecture is realized by a plug-and-play, flexible *infrastructure IP* called System Policy Update Engine (SPATE) and standardized *security wrapper* architecture that enable communication between the target (microcontroller) IPs and SPATE. The architecture is augmented with a cloud-based repository designed to establish an interface for off-chip provisioning and communication with SPATE through standardized communication protocol together with cloud service for validating the firmware patches and

978-3-9819263-5-4/DATE21/©2021 EDAA

ensuring the compatibility of the resident firmware to the upgrades. Fig. 1 shows a high-level overview of the system. CASTLE has the potential to significantly reduce turnaround times for firmware updates, streamline and in many cases obviate complex coordination necessary among various players in the supply chain in performing the updates, and ensure that the updates are authentic, trustworthy, and consistent with the target platform configurations.

The remainder of the paper is organized as follows. Section II provides the relevant background on challenges in firmware patching. We discuss the components of proposed framework in Section III. Section V illustrates our experimentation and demonstration plans. We discuss related work in Section VI and conclude in Section VII.

II. Challenges with Firmware Update

A major advantage of implementing system functionalities in firmware is in-field patchability. However, exploiting the benefits of in-field configurability requires a secure, disciplined, and reliable mechanism for firmware update which are not available in current industrial practice. On the other hand, due to the complexity and heterogeneity of the designs, firmware update today remains an ad hoc, and error-prone process, requiring significant coordination among a variety of players in a complex supply chain [6]. In particular, recall that firmware in one IP may need to interact with other IPs. Unfortunately, there is little coordination among different firmware vendors to ensure compatibility of the different firmware modules that co-habit any specific SoC design. The aggressive time-to-market requirements for a patch (particularly in presence of errors found in-field) makes it increasingly difficult to employ adequate validation time for different cross-compatibility scenarios. Consequently a firmware patch may be compatible with (or validated for) a specific variant of firmware running on different IPs. However, the very lack of co-ordination among different firmware vendors and complex, heterogeneous microcontroller architectures used in different IPs make it difficult to ensure that such compatibility issues are respected during in-field upgrades. Furthermore, the firmware load protocols themselves can be complex and can be subverted by an attacker to launch masquerading or time-of-check vs time-of-use (TOCTOU) attacks to load buggy or compromised firmware and disrupt system functionality [7]. Given that a subtle error in a firmware update can potentially make the entire system unusable, customers in field often postpone updates for a long time. This in turn has significant repercussions for the security of the system since a significant component of firmware updates address security vulnerabilities.

III. CASTLE Overview

The CASTLE infrastructure includes (1) an SoC security architecture that accounts for firmware updates from the ground up, and (2) a software architecture to enable coordination, verification, and unified delivery of firmware updates. The security architecture enables enforcing SoC-level firmware compatibility by defining the compatibility and TOCTOU requirements as security policies; The cloud-based software architecture is responsible for communicating with the SoC, delivering patches, and providing and maintaining compatibility constraints.

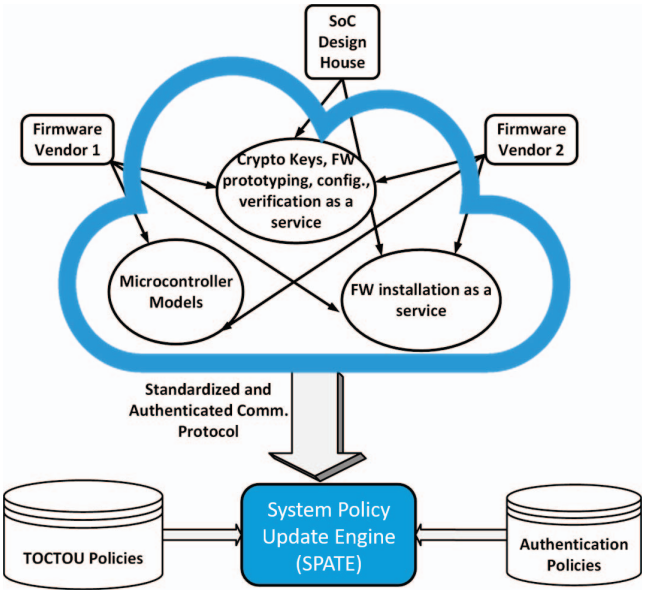

Fig. 1. A high-level overview of proposed infrastructure: (1) a centralized SoC security architecture, and (2) a cloud-based software service for firmware management.

A. SoC Architecture

Our key observation in the SoC architecture is that firmware load and patch protocol can be effectively implemented as security policies enforced through a centralized plug-n-play infrastructure IP which communicates with the different microcontrollers through a standardized communication interface. The following design components together provide the architectural support for systematic firmware load and patch and seamless configurability in field.

1) System Policy Update Engine (SPATE): The System Policy Update Engine (SPATE) is an infrastructure IP which is responsible for receiving a firmware update, authenticating the updated firmware through communication with the cryptographic engine, and communicating with the target microcontroller to load the firmware. This block acts as the centralized *patching agent* of the SoC. Fig. 4 shows how SPATE can be integrated to SoC designs as a stand-alone infrastructure IP. It communicates with the target microcontrollers through standardized security wrappers integrated to the IPs and ensure disciplined firmware update. Furthermore, SPATE ensures that the updates account for time-of-check time-of-use (TOCTOU) constraints, i.e., ensure that the firmware authenticated is actually the firmware loaded. We are augmenting the defined communication protocol among SPATE, cryptographic engine, and target IP, together with a formal proof of correctness with respect to TOCTOU properties. SPATE itself needs to be upgraded, in response to changing in-field requirements. To

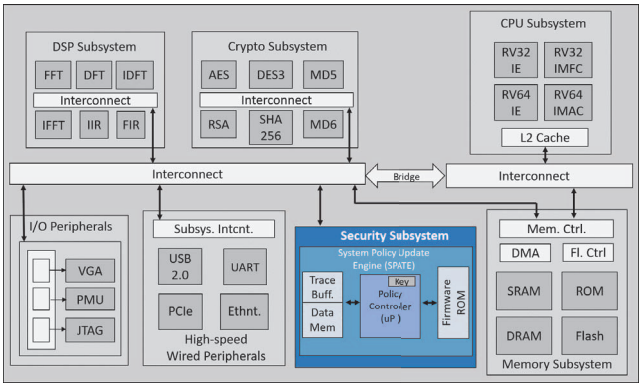

Fig. 2. An overview of SPATE in an illustrative SoC model.

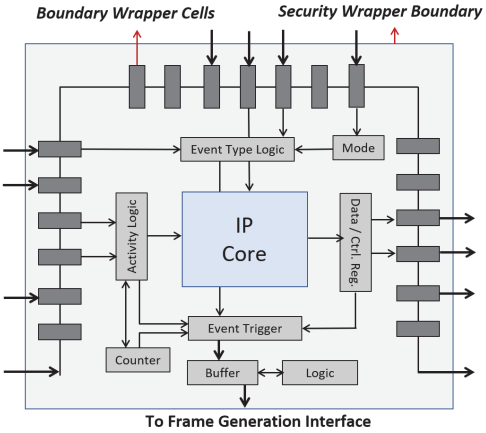

Fig. 3. Architecture of smart security wrappers.

address this, we designed SPATE itself as a micro-controlled IP running firmware to realize the functionality above. This enables it to be updated in-field as well. The architecture of SPATE is illustrated in Fig. 4. Formal verification of SPATE functionality is tractable because of the centralized nature: any analysis needs to explore only the SPATE design and its interfaces, not the entire SoC design [5].

2) Smart Security Wrappers: To implement the authenticated load and patch protocol, SPATE must *talk* to target microcontrollers and relevant peripherals like cryptographic engines. To facilitate these communications, we are extending the IPs that include microcontrollers with *smart* security wrappers responsible for communicating with SPATE. The smart security wrappers essentially are an extension of the test (*e.g.*, IEEE 1500 boundary scan based wrappers) which are universally available for functional verification. Fig. 3 depicts the architecture of the security wrapper. The wrappers are programmable, so that they can be configured at boot time as per patching requirements. Also, to enable smooth communication of SPATE with the IPs for this purpose without incurring additional overhead, we are investigating the possibility of repurposing

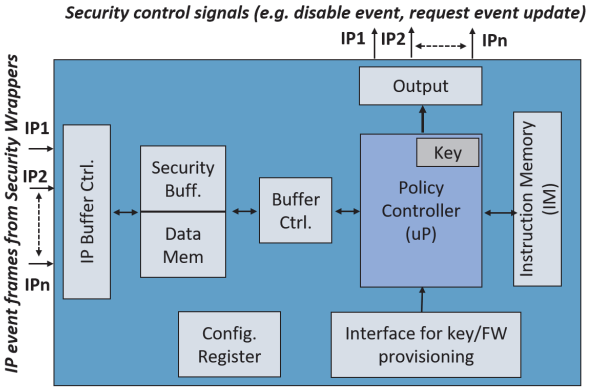

Fig. 4. Architecture of proposed System Policy Update Engine (SPATE).

an available communication fabric, possibly the debug fabric. This repurposing must be done carefully to ensure minimal disruption to the normal workload.

B. Software Architecture for Firmware Configuration Management and Delivery

The software architecture tackles the problem of managing and validating firmware updates, and communicating the updates to target SoC designs and platforms. A key challenge is that firmware modules for different IPs in an SoC might be managed by different vendors (who are responsible for the target IP) rather than the SoC integration house. To facilitate a standardized infrastructure usable by the various vendors, we propose a cloud-based repository that enables different vendors to manage their own firmware with relative isolation (Fig. 1). Our repository is modeled analogous to the github.com site that enables various firmware vendors to manage their updates, while providing a uniform, centralized pathway to securely communicate with the target systems. The role of SPATE as a centralized hub in SoC designs for managing firmware updates implies that the communication protocol for firmware delivery can be a single, uniform mechanism for communicating with SPATE. A key constraint with a firmware update is that it may only be compatible with (or has been validated against) specific versions of other firmware executing in other micro-controllers. The communication protocol accounts for the compatibility of the firmware during update. The proposed software system provides verification-as-a-service to enable firmware vendors to verify their updates. The cloud service includes virtual prototype models of several standard microcontrollers (e.g., MinuteIA, 8051, RISC-V), which can be used to validate various firmware configurations. Vendors can also use their custom microcontrollers or extend the ones available with environmental models for the surrounding custom hardware interface as necessary for validating their firmware.

C. Authentication and Remote Patch

Any update to SPATE must also be authenticated. To enable this, we are designing a bootstrapping load mechanism

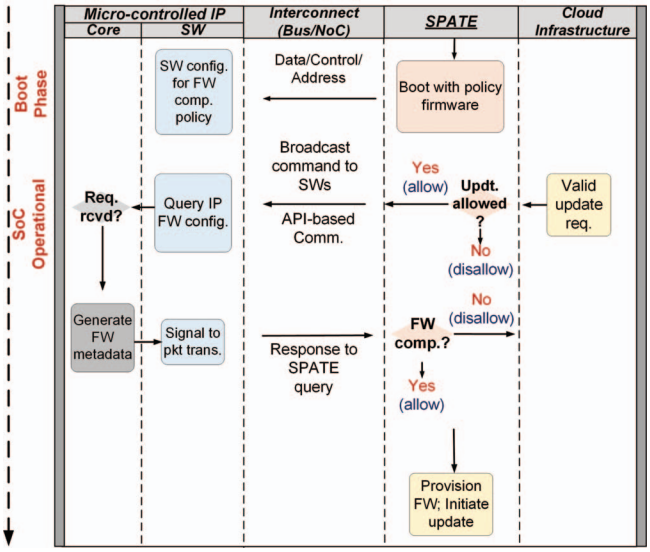

Fig. 5. Message flow diagram of proposed solution for ensuring FW compatibility.

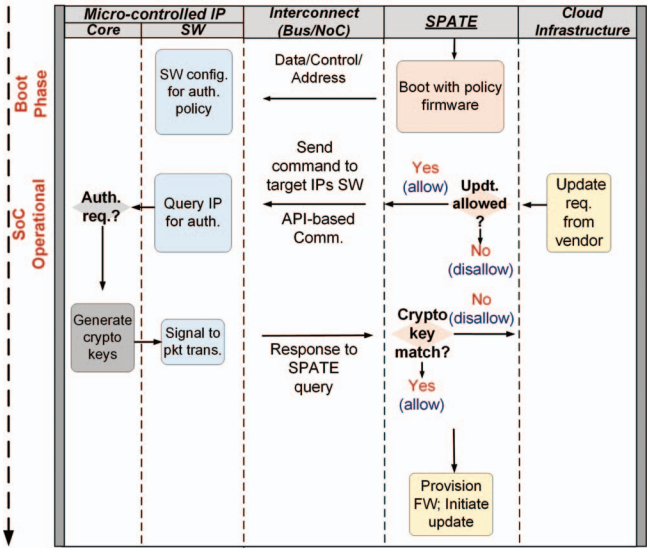

Fig. 6. Message flow diagram of proposed solution for system resiliency against masquerading attack.

with a provable guarantee of authenticated load. To protect the firmware patching process against attacks like malicious backdoors and Trojans, the framework incorporates an authentication mechanism based on cryptographic keys. SRAM-PUF is a suitable candidate for such key generation as it exploits the intrinsic process variations at power-up. The power-up key generation at secure boot prevents any on-chip key storage access control attacks. Also, the design of such weak PUF is cost efficient and doesn't require additional circuitry for on-chip implementation.

IV. Illustrative Examples

Firmware assurance can be established through a collection of fine-grained security policies implemented in CASTLE. These policies provide protection against a variety of attack scenarios that can violate the integrity, availability, and confidentiality requirements of the SoC platform. Here we discuss three representative policies corresponding to the functional and security flaws of firmware patching. These policies are outlined as illustrative examples to show how the proposed framework can help the security architects implement diverse set of security policies for firmware assurance.

A. Functional Flaw: Incompatible Firmware Update

Following is a typical requirement of firmware update in SoC platforms with micro-controlled IPs.

- Representative Policy: Any firmware patch must meet the compatibility requirements of the target platform.

Attack Scenario: We consider a scenario involving an attempt to update a micro-controlled IP with a firmware version that is incompatible to existing firmware of other IPs. This can be a malicious attack to impair the functionality of the platform or an inadvertent error due to lack of systematic approach in the patching mechanism.

Flow of operation: Fig. 5 shows the flow of events for a security policy to protect against such attack.

- At boot phase, SPATE configures the SWs according to policy specifications. A standardized, API-based communication between centralized SPATE and distributed SWs enable a secure, systematic way to accomplish tasks associated with policy enforcement.
- Upon receiving a firmware update request from the cloud infrastructure, SPATE broadcasts queries to the SWs located at the micro-controlled IPs. The SWs translates the bus packets and delivers signals to the target IP interface.
- The target IP responds to the query by generating firmware metadata such as firmware version, current configuration, compatibility requirements, micro-controller model, etc. The SW located at the target IP translates the data to packets and forwards to SPATE for verification.
- The execution core at SPATE validates the firmware metadata received from all micro-controlled IPs to detect any violation of compatibility requirements. In absence of possible violation, it responds to the query of cloud infrastructure and initiates firmware provisioning. In case of compatibility mismatch, SPATE discards the update request placed by cloud service by sending an error message.

B. Security Flaw: Masquerading Attack

This section shows how policy-based solution can be employed for masquerade prevention via proposed framework. The following is a typical requirement against masquerading attack:

- Representative Policy: Firmware update must comply with source authentication and authorization specifications.

Attack Scenario: We consider a malicious entity pretending to be an authentic firmware vendor and attempt to update the

target IP with malicious firmware. Failure to thwart such attack can lead to severe security breach and complete operational failure.

Flow of operation: The message flow diagram for the use case scenario with corresponding policy implementation is depicted in Fig. 6. The key steps are as following:

- At boot phase, SPATE configures the SWs according to policy specifications. Successful authentication and authorization for firmware update can be accomplished by several means based on the use case and security requirements of the target platform including PUF-based challenge response pairs, composite watermark hash, digital signatures, etc. The flexible architecture of the SWs can be augmented to facilitate such authentication mechanisms of the target IP.
- When an update request is placed by a firmware vendor from the cloud infrastructure, SPATE sends queries to the target IP to retrieve the authentication data.
- The target IP responds to the query by generating the crypto key (*e.g.*, PUF response, digital signature, watermark hash, etc.) and SW forwards the packets to SPATE.
- SPATE verifies the keys retrieved from the target IP with the ones received from the vendor requesting an update. A failed authentication leads to cancellation of the update request and a raised flag about the potentially spurious vendor.

C. Security Flaw: TOCTOU Attack

Here we show how the proposed security architecture can help to implement policies against TOCTOU attacks. A typical security requirement against TOCTOU attack is the following:

- Representative Policy: Overlapping firmware updates are prohibited.

Attack Scenario: In this scenario, the attacker attempts to exploit the race condition of TOCTOU and update the target IP with malicious firmware. This attack can lead to installation of compromised / illegal firmware on the platform.

Flow of operation: Fig. 7 illustrates the message flow diagram of the TOCTOU attack with implemented policy. The operational flow can described as follows:

- At boot phase, SPATE configures the SWs according to policy specifications. TOCTOU attack prevention policy can incorporate several aspects including single-threaded firmware update, barring all operations of target IPs during update, limiting / prohibiting access to the target IP during update, etc. In this case, we employ the SW located at the target IP to prevent all read / write request originating from other IPs and use SPATE to enforce a single-threaded firmware update.
- SPATE sends commands to the SW of the target IP to change the status from *operational* to *update* mode when a valid request is placed by an authentic vendor from cloud infrastructure.
- Once the SW responds back with change of IP status, SPATE initiates a non-overlapping firmware update process with the provisioned firmware. The single-threaded nature of the update eliminates any possibility of race condition introduced by overlapping update requests.
- Upon completion of the update process, the SW changes the mode of the IP to *operational*, sends acknowledgement to SPATE, and enables access requests to the target IP.

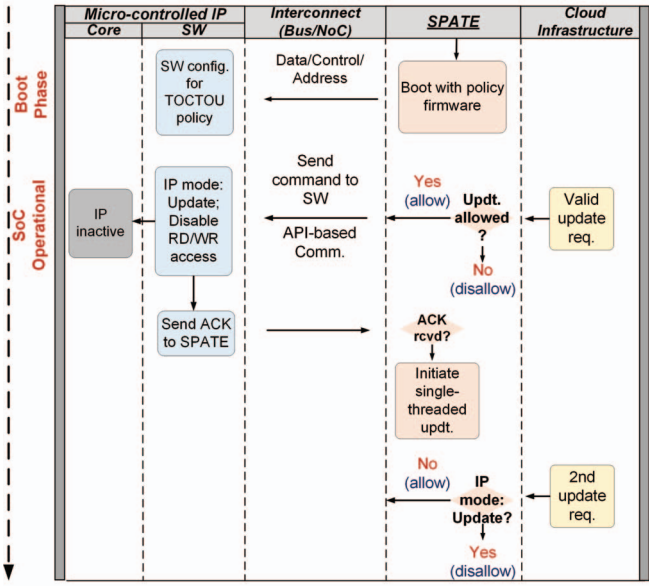

Fig. 7. Message flow diagram of proposed solution for system resiliency against TOCTOU attack.

V. Experimental Plan

Given the lack of open-source SoC models with sufficient complexity, we have been building our own SoC model, `AutoSoC`. It represents an academic SoC architected with all the major SoC components and incorporates many relevant design aspects of an industrial SoC. The architecture of `AutoSoC` is inspired by commercial automotive SoC designs with on-chip network fabric and application specific subsystems for multi-functionality. The IPs of each sub-system are clustered together based on their functionality. The CPU subsystem has 4 different RISC-V cores that support a variety of ISAs. The crypto subsystem incorporates a diverse set of crypto engines including AES, DES3, RSA, MD5, and SHA. The DSP subsystem is consisted of FIR, IIR, DFT, IDFT, and FFT block. Two RAMs, and a ROM are integrated as the memory components of the design. The design incorporates an Ethernet controller, an SPI modules, an UART, and a toy GPS model for external connectivity. The major interconnect of the SoC is an open-source, NoC, namely, LisNoC. It is compatible to wormhole-based flow control and has network adapters that support DMA engines and message passing functionalities. All IPs of `AutoSoC` are Wishbone-bus compatible and obtained from open-source repositories. `AutoSoC` is functionally validated in ModelSim.

Currently, We are augmenting `AutoSoC` with the SPATE unit, implemented as a micro-controlled soft IP. We are extending and adapting this model to function as a demonstration plat-

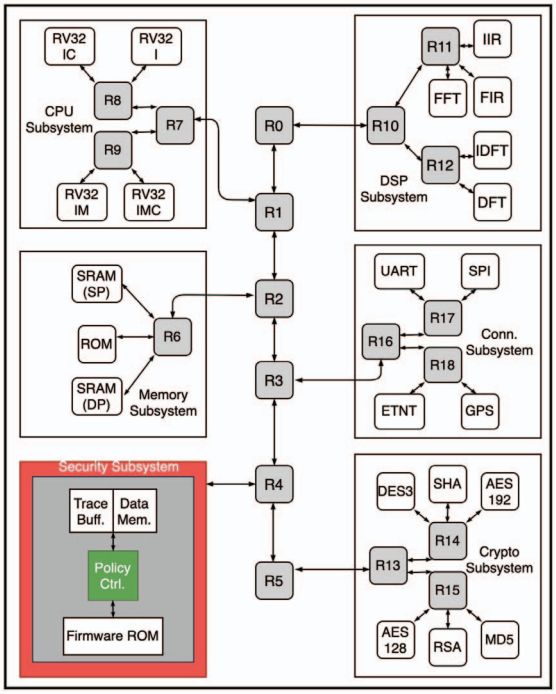

Fig. 8. The AutoSoC SoC Design

form. In particular, we are replacing several custom hardware IPs with microcontroller implementations using either 8051, RISC-V based microcontrollers, and minute-IA (if available), with associated firmware. We are modeling the firmware designs to reflect these industrial complexities. For the software infrastructure, we are exploiting a standardized open cloud infrastructure, and add virtual prototype models for standard micro-controllers as part of verification-as-a-service. The architecture serves as a proof-of-concept as well as a working prototype to be adapted/incorporated into an industrially viable open-source firmware management infrastructure.

VI. Related Work

There has been work in recent years on various aspects of firmware design and analysis, as well as validation of firmware load protocols. For instance, Krstic *et al.* developed an approach to verify firmware load protocol by specifying the communication between the IPs [7]. Sastry *et al.* [8] discussed a method for enforcing access control to security critical assets of SoC. Conti *et al.* [9] proposed a methodology to implement context-related policies for android application. However, these efforts do not provide a coordinated, unified mechanism for collection, analysis, and installation of updates. There are mechanisms for over-the-air (OTA) firmware updates, particularly for SoCs targeted towards applications with long field life *e.g.*, automotive, infrastructure, etc. [10], [11]. But these mechanisms vary depending on the microcontroller and firmware vendor. The design of SPATE is an adaptation of our previous work on systematically implementing SoC security policies [2]–[5], [12], [13]. That work targeted to achieve flexibility in security policies implementation and in-field hardware reconfigurability but did not specifically consider firmware patch.

VII. Conclusion

The key take-away from our work is that it is essential and entirely viable to develop systematic architectural support for ensuring runtime resilience of SoC designs against buggy or compromised firmware. In particular, this can be achieved by turning the resiliency requirement to the problem of security policy enforcement and developing (or re-purposing) policy enforcement architectures to implement such requirements. We showed how to do this for firmware patching, resulting in (to our knowedge) the first comprensive framework for disciplined, in-field firmware updates. The proposed infrastructure facilitates firmware patching in complex, heterogeneous SoCs comprising embedded micocontrollers of various architectures and service providers. Our work streamlines and simplifies complex coordination needs among firmware vendors and incurs minimal overhead by employing architectural artifacts that are readily available on-chip. Distinct features like flexible in-field configurability, and low overhead make this patching framework highly suitable for diverse applications in automotive, IoT, and infrastructure domains.

In future work, we will demonstrate the feasibility of proposed framework on our SoC model that is capable of reflecting the industrial complexities of firmware patching and consider extending the approach for SoC resiliency against other embedded software compromises.

References

[1] S. Ray and Y. Jin, "Security Policy Enforcement in Modern SoC Designs," in *IEEE ICCAD*, 2015.
[2] A. Basak, S. Bhunia, and S. Ray, "A Flexible Architecture for Systematic Implementation of SoC Security Policies," in *IEEE ICCAD*, 2015.
[3] ——, "Exploiting Design-for-Debug for Flexible SoC Security Architecture," in *DAC*, 2016.
[4] A. Basak, S. Bhunia, T. Tkacik, and S. Ray, "Security assurance for system-on-chip designs with untrusted ips," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1515–1528, 2017.
[5] A. P. D. Nath, S. Ray, A. Basak, and S. Bhunia, "System-on-chip security architecture and cad framework for hardware patch," in *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*. IEEE Press, 2018, pp. 733–738.
[6] J. Grundy, "Firmware validation: challenges and opportunities." in *FMCAD*, 2013, p. 11.
[7] S. Krstić, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor, "Security of soc firmware load protocols," in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2014, pp. 70–75.
[8] M. R. Sastry, I. T. Schoinas, and D. M. Cermak, "Method for enforcing resource access control in computer systems," Jul. 22 2014, uS Patent 8,789,170.
[9] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "Crêpe: A system for enforcing fine-grained context-related policies on android," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 5, pp. 1426–1438, 2012.
[10] P. L. Skan, "Method for over-the-air firmware update of nand flash memory based mobile devices," Apr. 13 2010, uS Patent 7,698,698.
[11] M. Shavit, A. Gryc, and R. Miucic, "Firmware update over the air (fota) for automotive industry," SAE Technical Paper, Tech. Rep., 2007.
[12] S. Ray, A. Basak, and S. Bhunia, "Patching the internet of things," *IEEE Spectrum*, vol. 54, no. 11, pp. 30–35, 2017.
[13] A. P. Deb Nath, S. Boddupalli, S. Bhunia, and S. Ray, "Resilient system-on-chip designs with noc fabrics," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2808–2823, 2020.