SSEL: An Extensible Specification Language for SoC Security

Kshitij Raj, Arrush Hegde, Atul Prasad Deb Nath, Swarup Bhunia and Sandip Ray

ECE Department, University of Florida, Gainesville, FL, USA
kshitijraj@ufl.edu, arrush.hegde@ufl.edu, atulprasad@ufl.edu, swarup@ece.ufl.edu, sandip@ece.ufl.edu

***Abstract*—A critical component of System-on-Chip (SoC) design entails specification of security requirements for trustworthy system operation. In this paper, we address this problem through the design of a new Security Specification Language (SSEL) to standardize the definition and implementation of SoC security requirements. We demonstrate that a viable path to the development of such a language is to build it on top of an existing programming language, extending a subset of the underlying language constructs and introducing new security-specific constructs as APIs. SSEL realizes this idea by building on top of a subset of C. We show how SSEL constructs can be used to develop executable specifications of a diverse set of usecases in modern SoC platforms. We show the application of SSEL in designing specifications for unlocking Logic Locked (LL)IPs, Authentication, and Secure Boot.**

I. Introduction

Modern System-on-Chip (SoC) designs include a variety of security-sensitive events, *e.g.*, distribution and disbursement of a variety of cryptographic keys, locking and unlocking IPs, authenticating inter-IP communications, etc. It is obviously crucial to ensure that such activities are implemented as intended by the security architects. Unfortunately, infrastructures for SoC security specification are lacking. In current industrial practice, SoC security specifications are defined in an ad hoc manner by different players at different stages in the SoC design life cycle, *e.g.*, it is common for security architects to provide high-level security assessment and architecture which is refined and modified by system integrators during various stages of design and implementation. To exacerbate the situation, security specifications in practice are still primarily captured in ambiguous English, together with informal charts, diagrams, and tables, and many requirements remain undocumented [1]. Unsurprisingly, "innocent" optimizations made with an inaccurate mental picture can result in subtle vulnerabilities that can compromise the entire system. Furthermore, if a vulnerability is detected late, *e.g.*, at post-silicon or in-field stages, then it can be expensive to fix and often results in brittle patches and point fixes often with cascading security effects.

In this paper, we develop a language, SSEL, for defining security specifications of modern SoC designs. The key idea of SSEL is to provide constructs for intuitive definition of inter-IP communications and usage scenarios targeting security protection against hardware attacks *e.g.*, logic locking, authentication, IP watermarking, etc. A key target of SSEL is to enable specifications providing protection mechanisms in SoC designs against a variety of supply chain adversaries and enabling systematic comprehension of security implications of different system-level communications. We demonstrate SSEL in the specification of several illustrative case studies inspired from the realistic SoC designs in the open-source community.

There has been recent work on designing a variety of specification languages to capture security requirements for microarchitectures and SoCs [2], [3], [4]. However, there has been little adoption of formal security specification languages in current industrial practice. A key reason for the lack of adoption of formal specification frameworks is the need for security architects to (1) learn unfamiliar specification paradigms and formalisms, and (2) define system behaviors that account for low-level implementation details [5]. SSEL addresses this problem by taking a fundamentally different approach to the specification paradigm. Most related specification languages target a succinct formalism for the language semantics typically based on some flavor of Temporal Logic.[1] The focus of SSEL, on the other hand, is *usability* by architects who may not have previous expertise in Temporal Logics. The key insight is that architects, albeit unfamiliar with formal logic and program semantics are indeed familiar with executable specifications and programming languages, *e.g.*, many architectural performance models use SystemC-TLM. Consequently, a language that extends a subset of a known programming language with specific constructs for intuitively capturing security-specific events is a viable approach for developing executable security specifications.

Fig. 1 represents the SoC lifecycle along with various security assessments during each life cycle and how we plan to integrate SSEL in the modern SoC design flow. In the current state of development, SSEL supports generating specifications using its formalized grammar. SSEL exploits the above insight as follows. Rather than developing a new formal grammar from scratch, the language is constructed by developing security constructs on top of a small but usable subset of the C language. An architect can simply treat the additional SSEL constructs as *Security APIs*. As we show in our case studies, an architect writing a specification in SSEL

[1]Since many security specifications focus on information flow, security languages generally support some form of hyperproperties that involve multiple copies of the state machines. However, the language semantics are still primarily influenced by Temporal Logics.

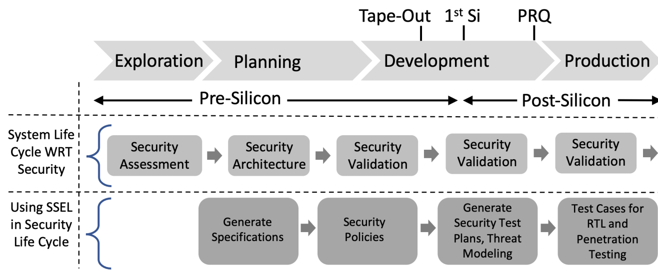

Figure 1. SoC System Life Cycle with Security Assessment using SSEL

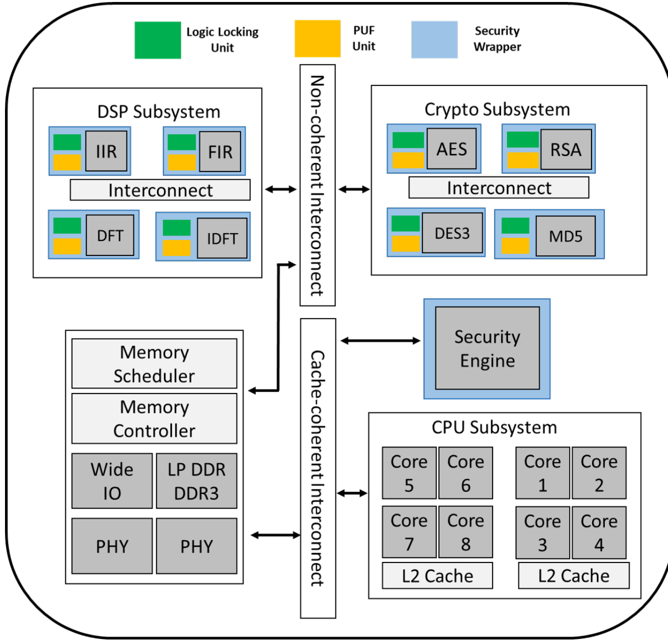

Figure 2. An illustrative SoC platform with architectural support for state-of-the-art security features

can essentially transcribe the security events from the diagrams and flows that they normally develop in current practice. In addition to providing a familiar environment for developing scenarios, the advantage of this approach is *extensibility*: in order to support a new security event, we simply need to develop an "implementation" of the event in C and provide the interface to the implementation as API. Finally, this approach requires no new tools to synthesis paradigms to develop an executable specification or exploration: one can simply leverage the existing C-based tools.

II. Background: An SoC Design with Illustrative Security Countermeasures

Fig. 2 shows an illustrative SoC design with some state-of-the-art security countermeasures for supply chain adversaries. While this specific model is influenced by (and a simplified, sanitized version of) realistic SoCs, the basic features are available in many commercial SoC designs as well. In particular, a key aspect of the SoC is the use of a centralized *Security Engine* that controls the security aspects of the whole system, which communicates with the IPs in the SoC through a variety of *Security Wrappers*. We discuss these features briefly here, to provide the flavor of security events that must be captured by a security specification language.

Security Engine: The Security Engine (SE) is responsible for the security events in the SoC. From an integration perspective, the SE is simply a subsystem that coordinates through a communication fabric. Internally, the SE includes a microcontroller running firmware which enforces the computation and communication required for the security coordination. The functionality implemented in the microcontroller depend on the security policies that need to be enforced in the SoC [6]. For the purpose of this paper, we assume that the security policies include specific boot policies that require IPs procured from an untrusted source to be unlocked and authenticated before they become functional.

Security Wrapper: Distributed security wrappers are deployed at IPs to enable the trusted microcontroller to perform security specific operations via assertion of security critical signals. The security wrappers are typically augmented implementation of IEEE P1500 test wrappers and standardized debug wrappers (*e.g.*, ARM®Coresight™) which come standard with most modern IPs to facilitate SoC debugging.

Physically Unclonable Function (PUF) based Authentication: The PUFs are implemented as distributed units in the security wrappers at IPs and the response extracted from the distributed PUF units are compared with golden reference signature to verify the authenticity of the IP. The typical authentication flow includes initiating the authentication process by sending challenge vectors to IPs, receiving the responses, comparing obtained CRPs with golden values, and updating IP status after authentication successes and failures, etc.

Logic Locking (LL): Additional hardware is inserted in the form of finite state machines (FSMs) or key gates to obfuscate the original functionality of the design. A design is "unlocked" (*i.e.*, operates normally) only after applying appropriate keys to the I/O ports of IPs. A common use cases of unlocking is system start-up as part of secure boot.

III. Overview of SSEL

A. The C Foundation and Extensions

In SSEL, we used a set of existing constructs from C and *loops*, *functions*, *structures*, and *data types* are used to define the constructs. We augmented SSEL with new *data types* to account for the binary and hexadecimal nature of the data transferred during on-chip transactions. Since SSEL is designed to provide abstraction of system-level communication, the technical details of various bus protocol implementations are encapsulated into APIs "baked" into the language abstraction. All inter-IP interactions are oblivious to existing bus implementations to enable reusability of SSEL specifications across platforms and interconnect protocols.

Using C as the baseline language for SSEL has it's own sets of limitations. Though it is convenient to bridge the gap between RTL designs and firmware code using C compared

Table I
DATA TYPES IN SSEL

Type	Storage Size	Value Range
bool	1-bit	0 (false) or 1 (true)
char	1-byte	-128 to 127
int	4-bytes	-2,147,483,648 to 2,147,483,647
long	8-bytes	-9223372036854775808 to 9223372036854775807
float	4-bytes	3.4E +/- 38 (7 digits)

to other high-level languages, there are some essential features required in security specification that cannot be directly supported in C. One main feature we added in SSEL is the usage of a *function* within a *function* for a set of built-in *functions*. Table 1 illustrates the primary *data types* supported by SSEL. We updated certain aspects of *functions*, *structures*, and *data types* in SSEL. Furthermore, we created several built-in *functions* for the ease of specifying generic use cases in SoC models, *e.g.*, SSEL supports two new data types *nodes* and *subnodes*.

B. SSEL *Constructs*

In addition to the existing C constructs, we developed a set of new constructs to facilitate inter IP communication for security enforcement in the SoC. These constructs are tailored to succinctly express IP-to-IP transactions and exploit the aforementioned C features to execute diverse security functions. Majority of the use cases requirements of standardized security measures can be specified via our constructs, some of which are enlisted in Table II.

Remark 1: The use of an off-the-shelf programming language as a foundation for SSEL means that it is relatively straightforward to develop a "compiler" that synthesizes an SoC security specification written in SSEL into executable code. Augmenting SSEL with a new construct simply entails providing an implementation template (in C) of new SSEL constructs, and integrate that implementation into the compiler. This enables us to re-purpose the compiler tool-chain already available for C to create executable code for SSEL. One upshot of this flow is a straightforward path to obtain *executable* specification of SoC security. These specifications can be used as golden reference models for the security implementation.

IV. SSEL CASE STUDIES

In this section, we demonstrate the usefulness of SSEL constructs to define representative security use cases. The case studies below are motivated by the SoC discussed in Section II. However, the SSEL specifications themselves are obviously independent of the SoC target. Note that although each case study can be implemented compactly with SSEL they are not trivial; indeed, RTL implementation of these features even on a simplistic SoC took several person months.

A. *Unlocking A Logic Locked IP*

Fig. 5 illustrates the message flow operations of unlocking a logic locked IP. Here, we describe the use case using the constructs in SSEL.

```
/* Node and SubNode Definition */
Node CCU(
   unlock = 1;
   bin Add = “”; Address as per specifications
) CCU;

SubNode key(
   bin Data = “” // Data = key
) key;
```

Figure 3. Node and Subnode Definition

```
/* Creating Message for SubNode */
Create(CCU, key, key_add, key_data, keym);
Message IP_Data_Name(
   bin key_add = CCU.Add;
   bin key_Data = key.data;
); keym

/* Defining Target Address */
Target IP_Name(
   bin IP_address = “”;
   bin target_address = “”;
); TargetVariable
Check IP_Name(CCU, IP, TargetVariable, Success);

/* Message Transfer to Target Address */
Send(CCU, key, sw, keym, Success, Keym1, keym1, bin,
Transfer_address);
Message Keym1(                  //New Message
   bin IP_Address = CCU.Add;
   bin IP_Data = keym.data;
); keym1
Transfer_Address = sw.Add // Saved Target Address

/* Receiving Message from Source */
Receive_boot(keym1, IP, Transfer_Address);
Message Keym1(
   bin IP_Address = CCU.Add; // Address Changed
   bin IP_Data = keym.data;  // New Message
); keym1
store(Key2, ip_stack, top);
```

Figure 4. Message Flow using SSEL Constructs

- We identify the nodes in the use case, *viz.*, the Security Engine and the IP security wrapper. The metadata for *nodes* for unlocking are the appropriate keys, which are encapsulated into *subnodes*. We convert the subnode into a *message* that can be only accessed by the Security Engine as the *owner*. Fig. 3 shows the relevant SSEL constructs.
- Note that the *send* function creates a copy of the *message*: the *ownership* of the original *message* belongs to the *node* that created the message. Fig. 4 shows how the message flow of keys and metadata from source to destination are achieved using SSEL constructs.
- The ownership of the copied *message* gets transferred from the source to destination *node* through the *receive* function. Once the unlocking key is received as the *message* data, it is stored in a stack of requests using

Table II
MAJOR SSEL CONSTRUCTS

Construct	Description
Node	The address of the IP is defined within the Node(binary variable) and the IP_Name is used to identify the node. A node can be locked or unlocked, but an unlocked IP can only be accessed by the trusted master IP.
SubNode	Subnode contains the data that an IP needs to transfer. The data stored in the subnode cannot be accessed directly. It can only be accessed via a message construct.
Message	Message maps the data from the Subnode to the address of the Node. Nodes with correct address can access the data in the message as the owner. A message can be created using the create function.
Target	Target node helps to identify the sender and receiver. It is used in tandem with check function to see if the message data is being sent to the right address.
Check	Check helps to verify the addresses while facilitating the transfer of message data. It takes two nodes and a target node as input and checks if the address in these nodes are correct or not.
Create	Create is used to create messages using a node and a subnode. The names of the message varies according to the parameter names.
Send	Send is used to send messages from one IP to another. The result of the check function helps verify the send function and the validity of the IPs involved in the transfer. A copy of it is created after checking the address. Transfer_address is equated to the address of the receiver IP after appropriate checks.
Store	Store saves message variables in a stack until its called upon. It uses a top function that points to the last data element in the stack. Whenever the stack is updated, the top variable is updated.
Remove	Remove is used to delete message variables from the stack once they are executed. The top variable is decremented with remove function but it is ensured that the stack is not empty.
Receive	Receive takes the output of the send function and checks if the address is valid. When the condition is true, the IP address of the message is changed to that of the receiver.
Receive_Boot	This variant of Receive used to send data during boot phase. The main difference in logic here is that it ignores the unlock variable in the node.
Unlock	Unlock is used to unlock an IP once the correct key is provided.
Init_Val	This construct has the definition of all memory-mapped registers, their offset, and their initialization values at reset.
Eth_Frame	The Ethernet frame construct enables packet formation in accordance with the IEEE 802.3 standard to send asset provisioning requests to the AMI.

the *store* function and consequently, the destination IP is unlocked using the *unlock* function.

B. Case Study on PUF-based Authentication

PUF-based authentication uses a challenge-response paradigm to authenticate an IP. Although functionally the mechanism is very different from unlocking, from a perspective of system-level coordination they are very similar. Correspondingly we can use the same definition "template" as follows. We use the same *node* and *subnode* definitions as unlocking locked IPs including the steps of initial address check. The critical component of the authentication is the communication and manipulation of the PUF response. Once the PUF challenge from the Security Engine reaches the destination IP, it follows the subsequent steps *i.e.*, defining the new target nodes, creating messages, and using the send and receive functions to complete transactions.

C. Secure Boot Flow

In the third case study, we consider a more elaborate feature, secure boot. The secure boot we show below has three phases. The specific activities are depicted in Fig. 7. This case study also showcases an important requirement for security specifications: the ability to compose lower-level flows into more complex scenarios, *e.g.*, secure boot uses logic locking and authentication as components. We note that composibility is a non-trivial feature for formal languages in general; however, in our case, we get it for free as a by-product of function composition in standard procedural programming.

Initialization Phase: Here, all associated peripherals with their initial state at power on. First, we identify the *nodes* to be utilized. The memory map registers use a SSEL construct called *init_val*, which defines the registers, the corresponding offset of the register, and their initialization values at reset.

Asset Provisioning Phase: This stage involves the Security Engine provisioning assets from the cloud. These assets include the golden vectors for PUF authentication and logic locking keys. The nodes of interest here are the Ethernet controller of the Security Engine and the Asset Management Infrastructure(AMI). SSEL has a dedicated construct for creating packets according to the Ethernet IEEE 802.3 frame, which is sent to the AMI to provision assets. The Ethernet packet has standard fields like Preamble/SFD, Destination and source address, Length, Data, and CRC. The Destination and Source Address is the AMI Cloud and the Security Engine Ethernet address. The data in the frame consists of the unique identifiers for IPs for which the asset is fetched. Based on the IDs, the AMI provisions assets to the Security Engine.

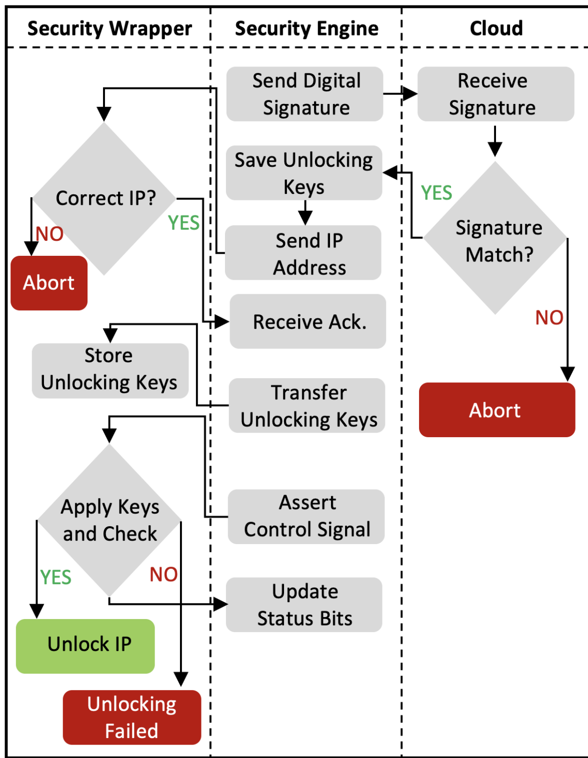

Figure 5. Message flow diagram of Unlocking a locked IP

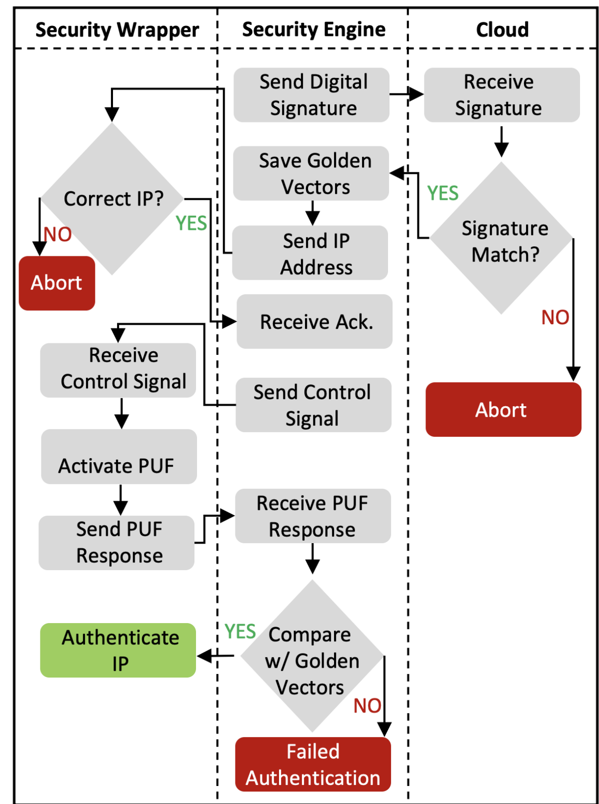

Figure 6. Message flow diagram of Authentication via integrated PUF

Security Operations Phase: Depending on the trust level of the vendor and manufacturer, each IP may be configured with a Security Wrapper to perform either authentication or logic locking, both, or neither. Based on the configuration, the Security Engine will orchestrate the specified security operations, including PUF-based authentication and IP unlocking. Once the sequence of operations is completed, the SoC can initiate normal operation.

V. Related Work

Design-for-Security (DFS) is commonly adopted in current industrial and academic practices to develop defense mechanisms against various attack surfaces. Prior research on DFS and SoC security architectures demonstrates methodologies for systematic implementation of security specifications [7]. The architectural approaches to cope with the challenges posed by heterogeneity and complexity of modern SoC designs include integration of dedicated infrastructure IPs and augmentation of standardized on-chip components such as test and debug wrappers for systematic implementation of security specifications [8]. However, the potency of such security architectures significantly relies on efficient definition of security specifications and standardized communication between the security components in the SoC design without interrupting it's regular operation. A formal, analyzable language such as SSEL can address such limitations and streamline the process of security feature integration via disciplined specification.

There has been significant research on methodologies for assuring secure RTL transactions and verification of SoC specification [9], [10], [11], [3], [4]. Some of these works include augmentation of existing languages with tailored expressions to define security use cases. Core Test Language (CTL) breaks event level transactions into digestible macros[10] to facilitate the implementation of security use cases. However, the application of CTL is design specific and dependent on certain test patterns. A methodology to formalize of language for SoC security specification is shown in [9]. The technical complexity of the approach, however, makes it difficult to execute. It is critical that the process of security specification definition is in complete compliance with original design flow for the ease of implementation [11], [12]. The challenges of modeling and verifying the behavior of heterogeneous SoCs have been alleviated in [3], [4] by formalizing the notion of Instruction Level Abstraction (ILA). These works provide a modular and uniform abstraction methodology to model the behavior of diverse of hardware accelerators and general purpose processors. The behavioral specifications of the SoC components is verified via equivalence checking. An approach to define the security features as hardware level policies and translating them to actionable design constraints is demonstrated in [6], [13]. These works develop security architectures for systematic implementation and verification of SoC security policies. Our work on SSEL is compliant with