# SeRFI: Secure Remote FPGA Initialization in an Untrusted Environment

Adam Duncan*, Adib Nahiyan†, Fahim Rahman†, Grant Skipper*
Martin Swany*, Andrew Lukefahr*, Farimah Farahmandi†, Mark Tehranipoor†
*Intelligent Systems Engineering, Indiana University, Bloomington, Indiana 47401 USA
†Electrical and Computer Engineering, University of Florida, Gainesville, Florida 32611 USA
Email: adamdunc@indiana.edu

***Abstract*—The bitstream inside a Field-Programmable Gate Array (FPGA) is often protected using an encryption key, acting as a root of trust and stored inside the FPGA, to defend against bitstream piracy, tampering, overproduction, and static-time reverse engineering. For cost savings and faster production, trusted system designers often rely on an untrusted system assembler to program the encryption key into the FPGA, focusing only the end-user-stage threats. However, providing the secret encryption key to an untrusted entity introduces additional threats, since access to this key can compromise the entire root of trust and breach the encrypted bitstream enabling a multitude of attacks including Trojan insertion, piracy and overproduction. To address this issue, we propose the Secure Remote FPGA Initialization (SeRFI) protocol to transmit the encryption key securely from a trusted system designer into an FPGA in physical possession of an untrusted system assembler. Our protocol eliminates direct key sharing with the untrusted system assembler as well as prevents against adversarial intention of extracting the encryption key during the programming phase where the assembler has physical access to the FPGA.**

***Keywords*—FPGA Security, Encryption, Secure Key Exchange**

## I. Introduction

Recent advancements in field-programmable gate array (FPGA) devices have enabled product designs ranging from low-cost consumer electronics to high-end commercial systems with reconfigurability, low development cost, and high performance [1]. The specific hardware functionality programmed into an FPGA is defined by a binary configuration file, called a **bitstream**, which is auto-generated via computer-aided design (CAD) tools by the designer. The bitstream file may contain sensitive and proprietary information and is often encrypted to ensure integrity and prevent intellectual property (IP) piracy [1, 2]. When encrypted, a **bitstream encryption key** is stored in non-volatile memory inside the FPGA so that the bitstream can later be decrypted during FPGA boot up. This encryption key, hence, serves as a root of trust for an FPGA-based system and must be protected accordingly.

Following the modern-day supply chain [3] for FPGA-systems, we consider the scenario where a **trusted system designer** designs a system requiring an FPGA to be procured, assembled, and programmed with a bitstream for the final product. To reduce both cost and production time, the designer often relies on a separate and potentially **untrusted system assembler** to purchase components and assemble the system, as well as physically shares the FPGA encryption key for programming the designer-provided bitstream into the FPGA. However, since the assembler is untrusted, or there may remain rogue employees in this untrusted environment, sharing this secret key allows compromising the root-of-trust. The adversary, with an access to the encryption key, can subsequently execute the following major attacks:

- reverse engineering proprietary bitstream for piracy [4].
- tampering with the bitstream and insert Trojans [5].
- reusing the key for system overproduction or cloning [1].

To prevent such threats, the system designer could use an in-house facility or an FPGA vendor [6] for key programming. However, both approaches increase the cost and the time-to-market with additional supply-chain complexity. Furthermore, an in-house approach is not often feasible for design houses that do not contain large-scale assembly and testing facilities.

In this work, we introduce the **Secure Remote FPGA Initialization (SeRFI)** protocol to securely and remotely load a secret encryption key into an FPGA without exposing the key to the untrusted assembler. Our protocol combines the cost-saving and time-saving benefits of using an untrusted assembler and adheres to traditional supply chain *without allowing any access to the encryption key*. Our protocol incorporates multi-party secure communication and integrity checking to perform key exchange from the designer directly to the FPGA while it is in the possession of untrusted the assembler during bitstream programming. To do so, a temporary tamper-resistant shared secret is created within the FPGA fabric and sent to the designer; which the designer uses to obfuscate and transmit a *partial bitstream* to be loaded at run-time that programs the actual encryption key into the FPGA non-volatile memory. Immediately after the encryption key programming, the shared secret and its means of regeneration are erased from the FPGA fabric resources to ensure confidentiality. We assume that once the key is physically programmed within the FPGA, it is protected from extraction [7]. SeRFI is augmented with capabilities to check against tampering with the partial bitstream and it is evaluated against rigorous attack models, assuming physical access by an untrusted assembler with state-of-the-art bitstream reverse engineering capabilities.

In this paper, we make the following contributions:

- We present SeRFI to allow secure remote FPGA encryption key programming for the first time without using external hardware security modules.
- We provide SeRFI with defenses against both bitstream-level and protocol-level attacks.
- We provide a security analysis to quantify the timing effort required by an attacker to bypass SeRFI.

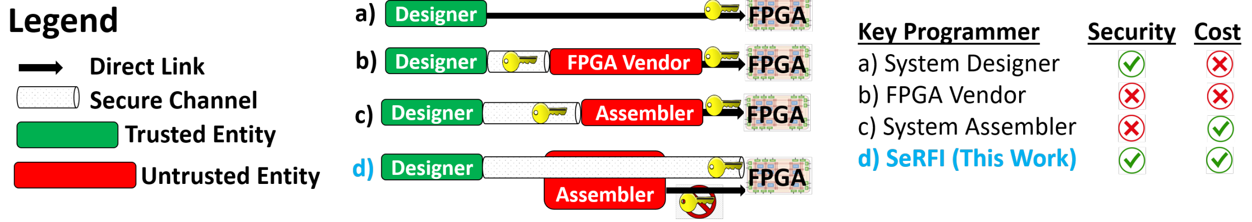


Fig. 1: (a)-(c) Current initial FPGA encryption key programming approaches are shown. d) Our Secure FPGA Remote Initialization (SeRFI) approach achieves high security at low cost.

The remainder of the paper is organized as follows. FPGA encryption key programming approaches and associated threats are discussed in Section II. The SeRFI protocol is introduced in Section III. SeRFI implementation steps and corresponding results are discussed in Section IV. Finally, we conclude our work in Section V.

## II. FPGA Remote Initialization

### A. Background

The standard approaches for the initial FPGA encryption key programming are shown in Figure 1. One possible approach is where the trusted designer programs the key at its secure facility, achieving high security at the expense of increased cost (see Figure 1(a)). Likewise, FPGA vendors such as Xilinx and Microsemi offer encryption key programming at affiliated facilities (see Figure 1(b)), again with an increased cost [8]. However, in this model, the encryption key needs to be shared with FPGA vendors. It also limits the usage of FPGAs. Finally, the designer can outsource the programming of the key to the untrusted assembler to reduce the cost at the possible expense of security (see Figure 1(c)). This is the most common trend for commercial products [3]. Our SeRFI approach, as shown in Figure 1(d), combines the low cost of utilizing the untrusted assembler for key programming with the security of denying the assembler direct key access.

Microsemi recently released its Secure Production Programming Solution (SSPS) for initial encryption key programming which requires external Thales hardware security modules (HSMs) [6]. SSPS uses the security features of external HSMs to securely transmit a key from the designer to the off-site FPGA. Unlike Microsemi's, our SeRFI approach requires no additional hardware modules. SeRFI also does not require a pre-existing factory-programmed key, allowing for multiple uses per FPGA. SeRFI is also applicable across all FPGA brands and configuration memory variants. Finally, attack surface is reduced by eliminating external HSMs as well as allowing user customization during SeRFI implementation.

### B. Threat Model

Our threat model assumes an untrusted system assembler that targets to obtain the encryption key to be loaded into the FPGA. Motivations for the attacker can be IP piracy, tampering, and overproduction as mentioned in Section I. We assume a strong attack model where the adversary has physical access to the FPGA and has extensive computational resources. In addition, we assume that the attacker can (and will) reverse engineer previous protocol captures in attempts to spoof various protocol components, and the system designer has no control over the offline activity of the assembler.

## III. SeRFI Protocol

Figure 2 shows a high-level overview of our proposed Secure Remote FPGA Initialization (SeRFI) protocol. Step 0 establishes a communication interface between the designer and the FPGA that is maintained for authentication, information, and key exchange. Steps 1 and 2 performs a multi-step authentication between the designer and FPGA to establish a temporary FPGA-unique shared secret $SS$ which acts as the security base for the following step. The final step uses $SS$ to obfuscate and transmit the actual encryption key from the designer to the FPGA, program the devices with the key, and lastly delete $SS$. In our proposed scheme, protections against both protocol attacks and physical attacks, such as FPGA input-output (IO) monitoring and bitstream tampering, are included to render any SeRFI attack moot, as it will require a complete bitstream reverse-engineering combined with a dynamic simulation component making attack times orders of magnitude larger than the protocol execution time. A complete version of the protocol is shown in Figure 3 detailing the operations performed by the designer and the assembler. The only hardware-specific blocks required for protocol implementation are for partial reconfiguration capability and run-time encryption key programming. These features are common on most Xilinx, Intel, and Microsemi FPGAs [8], [9], [10].

### A. Step 0 – Process Initialization and Key Creation

The protocol begins with the designer creating the master encryption key, $K_M$ intended to be loaded into the FPGA. The designer also estimates a minimum time value $t_{1max}$ that the untrusted assembler needs to perform a successful attack against SeRFI to act as a protocol decision point for potential abortion against any time-bound attack. ($t_{1max}$ estimation is discussed in Section IV.) An asymmetric key pair $\{K_A,K_B\}$ is also generated for the authentication process to be done in the following steps. A regular communication

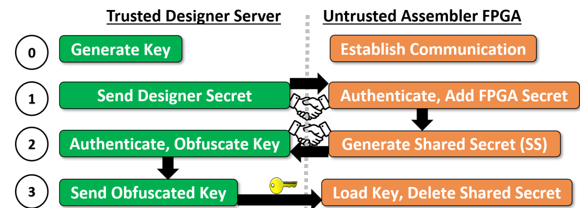


Fig. 2: A high-level conceptual view of the SeRFI protocol.

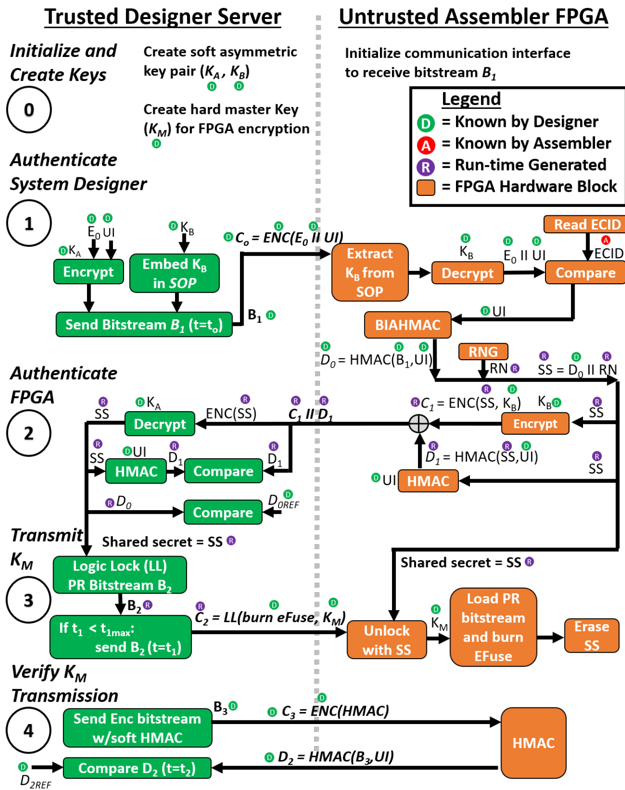


Fig. 3: A detailed view of the SeRFI protocol.

link (which may be insecure at this point), such as internet connectivity over Ethernet, is then established between the designer and the FPGA at the untrusted assembly. An optional plaintext bitstream $B_0$ incorporating the necessary networking infrastructure to communicate to the designer may be prepared and shared to the assembler to establish this link. Tampering with this channel for monitoring transmitted information does not reveal any sensitive information about the SeRFI protocol and underlying keys (as we will see in following sections), and, therefore, it is safe to share this design with untrusted assembler for easier implementation.

*B. Step 1 – Authenticating the System Designer*

In this step, the designer first targets a specific FPGA, identified by the device-specific electronic component chip ID (ECID) $E_0$, which is to be programmed using the SeRFI protocol. ECID is embedded into the device by the manufacturer, and may be collected from the assembler initially during procurement or during initialization at Step 0. Then, the designer generates a unique identifier nonce $UI$ for the target FPGA. $E_0$ and $UI$ are asymmetrically encrypted offline into ciphertext $C_0$ using $K_A$, and a plaintext bitstream $B_1$ is generated containing $C_0$ as well as $K_B$ embedded as a hardware-based stealthy opaque predicate (SOP) [11]. In general, the SOP mechanism can obfuscate constants values (i.e., the key $K_B$ for our case) within the bitstream using finite-state machine (FSM) encoding so that the value ($K_B$) cannot be obtained using standard static bitstream analysis techniques [12]. As shown in Figure 4, the next state logic is used to transition an FSM from its initial register values

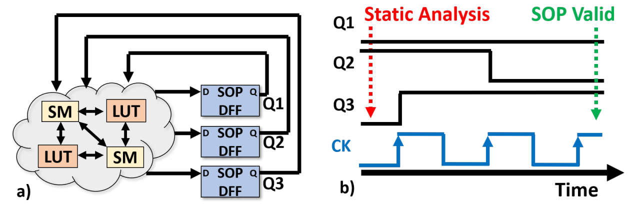


Fig. 4: a) Obfuscating constant inside the partial bitstream using hardware SOP implementation utilizing 3 DFFs. b) Example DFF state obfuscation by modifying LUT and switch matrix (SM) connections illustrating resistance to static analysis.

into a design-dependent future state. The register values at this future state are then used to provide $K_B$ within the hardware. The SOP concept thus forces the attacker to perform a reverse engineering *and* dynamic simulation to determine the register value at the specific time when it is interpreted as a ‘constant’ by the hardware. The reverse engineering required by the attacker to determine $K_B$ affects the time that the response needs to be sent back. Therefore, the designer can detect the breaching attempt to the protocol.

After this, $B_1$ is sent to the assembler at time $t_0$ to run on the target FPGA. When $B_1$ begins running, it unrolls $K_B$ from the SOP and uses $K_B$ to decrypt $C_0$ into $UI$ and $E_0$. The ECID embedded in the FPGA is next extracted and compared to a designer-known $E_0$. The ECID comparison is used to prevent an attacker from loading $B_1$ on a different FPGA. Upon successful ECID comparison, a hashed message authentication code (HMAC) operation using our custom built-in authentication HMAC (BIAHMAC) (see below) is initiated over the entire $B_1$ using $UI$ as a key to produce hash digest $D_0$. A random number generator (RNG) implemented within the FPGA fabric next produces a random number $RN$. A concatenation of $RN$ and $D_0$ is performed to create the shared secret $SS$.

To ensure that the untrusted assembly has not modified the bitstream $B_1$, we propose a custom BIAHMAC block embedded in $B_1$ that provides tamper detection and includes protection from reverse engineering and offline computation attacks. The basic BIAHMAC functionality is shown in Figure 5(a). We utilize the run-time configuration memory reading capability included in most current FPGAs, such as the Xilinx internal configuration access port (ICAP) [1], for BIAHMAC implementation. Here, an ICAP block reads the entire FPGA configuration memory at run-time while connected to an HMAC block used to calculate a running hash digest. The ICAP inputs are sourced by FSM to cycle through the different configuration memory address ranges. For example, a simple counter can be used to increment the address $Q_0 : Q_N$ and exhaust the complete address range. Any tampering to $B_1$, such as adding additional circuity to spoof $RN$ or to route information off-chip, would require a change to the FPGA configuration memory and, therefore, produce an incorrect $D_0$.

Our BIAHMAC includes additional protection mechanisms as shown in Figure 5(b). To prevent an adversary from performing a reverse engineering effort on one instance of $B_1$ and using it on future instances of $B_1$, we both randomize and obfuscate the *order* that configuration memory addresses are

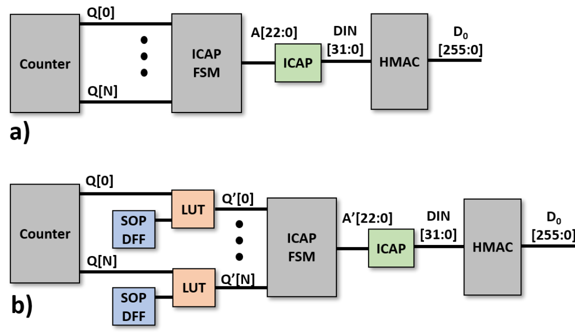


Fig. 5: a) HMAC performing run-time computation over entire FPGA fabric. b) BIAHMAC with added SOP structures so that the HMAC algorithm reads the fabric in a unique manner for every bitstream.

accessed with the ICAP. We use the previously discussed SOP concept to encode a constant in the hardware that is resistant to static bitstream analysis [11]. We then use this constant to determine whether or not each bit in our address counter is inverted. As a result, for $N$ counter bits, there exist $2^N$ permutations of address $Q'_0 : Q'_N$ accesses. For the 23-bit address range of a Xilinx 7-series frame address register (FAR), this corresponds to $8.4 * 10^6$ different possible combinations [9]. An attacker who reverse engineered an instance of $B_1$, is then unable to use this knowledge to accelerate the calculation on the next instance of $B_1$ as $Q'_0 : Q'_N$ will change for the subsequent programming and FPGA instances.

### C. Step 2 – FPGA Authentication

The FPGA, still programmed with $B_1$, next asymmetrically encrypts $SS$ using $K_B$ to produce ciphertext $C_1$. An HMAC signature $D_1$ is also generated for $SS$ using the key $UI$. $C_1$ is concatenated with $D_1$ and sent to the designer. Note that with asymmetric encryption, an attacker extracting $K_B$ from the bitstream is not able to decrypt $C_1$. Additionally, an attacker attempting to spoof $SS$, would not be able to extract $UI$ without tampering $B_1$, which would result in an incorrect $D_0$ calculated by the BIAHMAC in the previous step.

The designer uses $K_A$ to decrypt $C_1$ into $SS$ allowing for both the designer and FPGA to now have possession of the shared secret. An HMAC operation is conducted on $SS$ using key $UI$ to verify the authenticity of the FPGA by comparing $D_1$. Next, $D_0$ is extracted from the $SS$ and compared with the pre-computed reference $D_{0REF}$. If both comparisons are successful, the protocol continues to Step 3.

### D. Step 3 – Transmission of $K_M$ to FPGA

The designer next uses $SS$ to create a logic locked bitstream $B_2$ which contains functionality to program $K_M$ into the FPGA. We assume a lookup table (LUT)-based logic locking strategy similar to LUT-LocK [13] where $k$ bits of a logic locking key are pre-routed to $k$ pins of an $N$-input LUT, with $N$-$k$ LUT pins utilized to implement the functional design. Once $SS$ arrives at the designer, $SS$ is used to modify the LUT initialization values such that the logic only functions correctly if $SS$ exists in $B_1$. The designer implements this process by first placing the FPGA software at a checkpoint awaiting the logic lock key bits in the form of LUT initialization values.

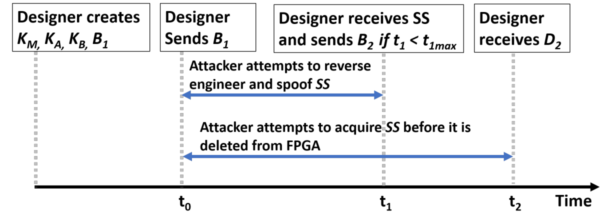


Fig. 6: The timeline for the SeRFI protocol.

The designer then evaluates the system time $t_1$, and sends $B_2$ to the assembler if $t_1 < t_{1max}$. If $t_1 \geq t_{1max}$, the protocol aborts, and returns to step 0 with complete regeneration of $K_A$, $K_B$, and $UI$.

The FPGA loads $B_2$ using partial reconfiguration and unlocks the logic locking with $SS$. Once unlocked, $B_2$ programs $K_M$ into the FPGA and subsequently deletes $SS$ from the FPGA memory space. At this point, the initial encryption key programming of the FPGA is complete and the attacker has no means of $K_M$ reconstruction.

### E. Step 4 – Verification of $K_M$ Transmission):

To validate $K_M$ programming, the designer next uses $K_M$ to create an encrypted bitstream $B_3$ with an HMAC in place to calculate a runtime hash digest of the configuration memory. $B_3$ is then sent to the FPGA, still in possession of the assembler. The FPGA loads $B_3$ which computes a run-time hash digest $D_2$ over $B_3$ using key $UI$ and sends $D_2$ to the designer. The designer compares $D_2$ to a pre-computed digest to verify that $K_M$ has been loaded correctly and marks this time as $t_2$.

### F. Protocol Timeline:

A timeline is presented in Figure 6 to illustrate the timing threshold decision points in the protocol. At $t_0$, $B_1$ is sent from the trusted designer and is used as a point of reference. The next checkpoint in the protocol occurs at $t_1$ when the designer evaluates whether $t_1$ is within the $t_{1max}$ threshold for the given attack model and determines whether to send the locked bitstream $B_2$. The final threshold occurs at $t_2$ where the designer receives and compares $D_2$. If $t_2$ is less than the threshold for the given attack model, the encryption key can be assumed as securely programmed. Otherwise, the process is aborted.

## IV. SeRFI Implementation

To evaluate the SeRFI protocol a 256-bit $K_M$ was chosen for loading into a mid-range Xilinx Artix-7 35T FPGA. Attack resilience was conducted using multiple attack vectors to determine attack time estimates.

### A. FPGA Resource Utilization

The primary components bitstreams $B_1$-$B_3$ are shown in Figure 7 and described with respect to each bitstream below. Table I illustrates their corresponding resource utilization and cycle counts. Utilization and cycle count data are obtained from a combination of our own synthesis and simulation results as well as results from the literature [14], [11]. Our FPGA resource utilization is seen to be $< 30$ % for each

TABLE I: Resource utilization for SeRFI FPGA implementation on a Xilinx Artix-7 35T device.

| Bitstream | Block | LUTs | FFs | % Slices | # Cycles |
|---|---|---|---|---|---|
| $B_1$ | RSA EncDec | 932 | 559 | 2.8 % | $3 * 10^6$ |
| $B_1$ | HW SOP | 256 | 1024 | 2.4 % | 32 |
| $B_1$ | BIA HMAC | 4054 | 2341 | 19.5 % | $9.6 * 10^5$ |
| $B_1$ | Soft HMAC | 1023 | 1022 | 4.9 % | 256 |
| $B_1$ | RNG | 90 | 32 | 0.4 % | 10000 |
| $B_2$ | HW Predicate | 256 | 256 | 1.5 % | 32 |
| $B_3$ | soft HMAC | 1023 | 1022 | 4.9 % | $9.4 * 10^5$ |
| *Combined* | - | - | - | - | $5 * 10^6$ |

of the three bitstreams. Combined cycle counts across all bitstreams are shown as $< 5*10^6$, resulting in a combined run-time execution time of $< 1$ s with a modest 100 MHz clock frequency. Specific details with regards to each bitstream are included below.

**Bitstream 1:** For asymmetric encryption and decryption estimation, a soft 1024-bit RSA implementation was chosen, which has been shown to fit into 557 slices [15]. SOP [11] structures to store a 1024-bit $K_B$ were estimated using 1024 DFFs to store the eventual constant, combined with 256 6-input LUTs to realize $256 * 2^6$ permutations of next-state logic for obfuscation. We designed, synthesized, and tested our custom BIAHMAC module to produce $D_0$ using a total of 1014 slices. Our BIAHMAC incorporated a SHA-3 open cores project [16], combined with our FSM and counter to drive the ICAP to incrementally access the entire configuration memory. We also used this SHA-3 core independently as an estimate for the HMAC producing $D_1$. The RNG hardware estimate to provide a 32-bit $RN$ was based upon the TI-TRNG paper [14] and thus estimated at 360 slices.

**Bitstream 2:** The logic locked bitstream $B_2$ includes locked gates that utilize $SS$ as the unlocking key. After unlocking, another SOP evaluation structure, with 256 DFFs, evaluates the 256-bit $K_M$. An FSM is also used to activate the fuse burning circuitry to burn $K_M$ into the on-chip eFuse. Lastly, circuitry is utilized to delete $SS$ from the fabric.

**Bitstream 3:** The encrypted bitstream $B_3$ includes just one primary hardware block which is another BIAHMAC instance to calculate the digest $D_2$. Note that since $K_M$ is known in advance by the designer, $B_3$ is already generated at the time $t_0$ and is ready to transmit as soon as the protocol allows.

### *B. Protocol Time Estimation*

Our protocol timeline starts with the transmission of $B_1$ including network traffic delays, followed by the designer parsing FPGA responses and applying logic locking to $B_2$, as well as run-time cycles of the FPGA executing $B_1$-$B_3$. *We provide a protocol time estimation of* 6 *seconds* in our example case study by examining the contribution of each component involved in SeRFI. Our estimates are discussed below and summarized in Table II.

**Step 1:** The first protocol step begins with the designer transmitting a pre-computed 3 Mb Artix-7 bitstream file $B_1$ to the assembler. We estimate this at 0.16 seconds using network transmission speed estimates of 19 Mbps, the slowest average upload and download rates for the top 100 countries [17]. We

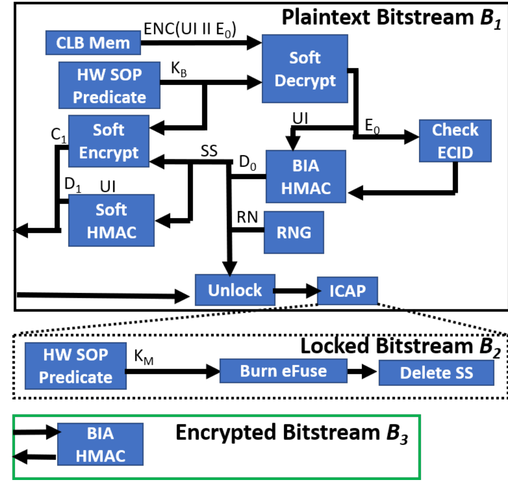


Fig. 7: The bitstream-level implementation of SeRFI.

next estimate the time to load $B_1$ onto the FPGA at 3.125 ms, assuming a 60 MHz configuration clock (CCLK) connected to 16-bit byte peripheral interface (BPI) as specified in Xilinx configuration documents [9]. Referring back to Table I, we bound our cycle count for $B_1$ during Step 1 at $10^6$ cycles, resulting in 4 ms of run-time execution.

**Step 2:** The second protocol step includes a run-time contribution from $B_1$, which we again conservatively estimate at 4 ms. A network delay is incurred transmitting $C_1$ and $D_1$, with size $< 1$ kB and bounded by 1 ms. There is also a delay incurred by the designer to decrypt $C_1$ and perform comparisons on $D_0$ and $D_1$, which we bound to 100 ms.

**Step 3:** This step accounts the time for the designer to parse the FPGA responses and lock $B_2$. We experimentally estimate this time contribution as around $5s$ by performing the bitstream generation step after a LUT initialization-value modification from a check-pointed state. It is performed on an Intel I5-8250U processor with 24 GB of RAM. We note the potential for speed increases during this step utilizing high-performance computing, as well as direct bitstream manipulation techniques [12]. Once $B_2$ has been created, it is sent across the network with an estimated 4 ms delay.

Once the FPGA receives $B_2$ it uses the ICAP with a 32-bit data bus to load $B_2$. The loading time is estimated at 1.6 ms with a 60 MHz ICAP clock with a 3 Mb $B_2$. After loading, $B_2$ requires approximately 51.2 ms to burn a 256-bit $K_M$ using 200 $\mu$s per fuse [18] time to blow estimates. To defend against side-channel attacks during the eFuse burning, randomized one-hot sequences of $K_M$ may be burned independently. For example, if $K_M[4:0] = 0101$, then sequences of 0001 and 0100 may be programmed in order to prevent simple power analysis attacks from monitoring the distances between current consumption spikes to infer specific programmed bits.

### *C. SeRFI Attack Resiliency*

We assume that an adversary can launch different attacks to circumvent our proposed SeRFI protocol with an ultimate goal of obtaining $K_M$. Potential attacks and respective in-built countermeasures are listed as follows.

- **Attack:** Since $K_M$ is never transmitted in the clear, the adversary must first reverse engineer $B_1$ and $B_2$

TABLE II: Time estimation for SeRFI FPGA implementation for Steps 1-3.

| Protocol Step | Protocol Action | Time (s) |
|---|---|---|
| 1 | network traffic: sending $B_1$ to assembler | 0.16 |
| 1 | load $B_1$ on FPGA | 0.003 |
| 1 | $B_1$ running on FPGA | 0.004 |
| 2 | $B_1$ running on FPGA | 0.004 |
| 2 | network traffic: sending $C_1 \parallel D_1$ to designer | 0.001 |
| 2 | designer decrypt $C_1$, compare $D_0$, $D_1$ | 0.1 |
| 3 | $B_2$ obfuscated by designer | 5 |
| 3 | network traffic: sending $B_2$ to assembler | 0.16 |
| 3 | load $B_2$ on FPGA | 0.003 |
| 3 | $B_2$ running on FPGA | 0.051 |
| $Complete$ | Complete SeRFI Protocol | $< 6$ |

to understand the construction of $K_M$, and learn its dependence on $SS$. Given enough time and resources within a single protocol session, an attacker can perform a detailed reverse engineering effort to determine the bitstream location of critical components such as $SS$, $RN$, $UI$, $D_0$, and $K_B$. However, this information cannot be applied towards a future session, since all these values, as well as their positions in the bitstream $B_1$, change from session to session. The attacker, therefore, must focus on attack vectors within a given session with attempts to extract $SS$ or spoof information sent from the FPGA to the designer.
**Countermeasure:** We estimate the minimum required time for a combined reverse engineering and simulation per session defined as $t_{1max}$. We include a decision point in SeRFI to abort if the designer has not received the correct information from the FPGA within $t_{1max}$ in Step 2. To quantify $t_{1max}$, we establish a lower bound by noting that state-of-the-art published bitstream reverse engineering tools [5] for Xilinx mid-range devices report an average of 300 *minutes* to produce a usable netlist. Furthermore, bitstream reverse engineering tools have not been published for Microsemi and Intel FPGAs. SeRFI protocol time estimation for a Xilinx mid-range device is $< 6s$ from Table II, or roughly **3000x** less than $t_{1max}$.

- **Attack:** An attacker can attempt to extract $SS$, or other critical run-time protocol values, by adding targeted logic (e.g., Trojans) to the bitstream $B_1$ for leaking the information through an FPGA I/O pin or other side-channels.
**Countermeasure:** Our proposed BIAHMAC can detect *any* tampering to $B_1$ by performing a run-time hash throughout the entire FPGA configuration memory to create $D_0$. Step 2 of SeRFI performs a comparison at the trusted designer's facility between $D_0$ and pre-calculated $D_{0ref}$ known only to the designer. If a miscompare is detected, $B_2$ is never sent to the attacker, and the $K_M$ is never exposed.
- **Attack:** An attacker may also attempt to spoof information sent to the designer in Step 2 in hope of creating a known $SS$ value to de-obfuscate $B_2$ outside of the FPGA. Here, the attacker chooses $RN$, and performs (exhaustive) computations outside of the FPGA to reconstruct $D_0$ and $D_1$ such that comparisons by the designer with pre-computed $D_{0ref}$ and $D_{1ref}$ are successful and a valid $B_2$ is sent to the attacker. The attacker can then use their known $RN$ to reconstruct $SS$, to de-obfuscate $B_2$ and expose $K_M$.
**Countermeasure:** We defend against this spoofing attack by enforcing $D_0$ and $D_1$ calculations to require a unique reverse engineering effort and dynamic simulation with *every* protocol session. SOP structure connections used to obfuscate $K_B$, as well as the algorithm used to access the fabric resources with our BIAHMAC block, are changed after each protocol session to defeat learning-based attacks and enforce the session-specific reverse engineering and simulation.

## V. Conclusion

In this paper, we presented the Secure Remote FPGA Initialization (SeRFI) protocol to securely program an initial encryption key into an FPGA through the use of an untrusted system assembler entity without requiring the use of any commercial hardware security modules. SeRFI includes tamper detection and protection mechanisms to defend against both protocol-level attacks and physical attacks on the FPGA. Protocol simulations estimate complete protocol execution times $< 6$ seconds, with adversarial attacks requiring upwards of 300 minutes.

## References


[1] S. M. Trimberger and J. J. Moore, “Fpga security: Motivations, features, and applications,” *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1248–1265, 2014.

[2] A. Lesea, “Ip security in fpgas,” *Xilinx http://direct. xilinx. com/bvdocs/whitepapers/wp261. pdf*, 2007.

[3] S. J. Mason, M. H. Cole, B. T. Ulrey, and L. Yan, “Improving electronics manufacturing supply chain agility through outsourcing,” *International Journal of Physical Distribution & Logistics Management*, vol. 32, no. 7, pp. 610–620, 2002.

[4] F. Benz, A. Seffrin, and S. A. Huss, “Bil: A tool-chain for bitstream reverse-engineering,” in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pp. 735–738, IEEE, 2012.

[5] T. Zhang, J. Wang, S. Guo, and Z. Chen, “A comprehensive fpga reverse engineering tool-chain: From bitstream to rtl code,” *IEEE Access*, vol. 7, pp. 38379–38389, 2019.

[6] Microsemi, “Secure production programming solution (spps) user guide,” *Secure Production Programming Solution (SPPS) User Guide v11.8 SP1*, 2018.

[7] Microsemi, “Ug0443,” *User Guide: SmartFusion2 and IGLOO2 FPGA Security and Best Practices v10.0*, 2019.

[8] Microsemi, “User guide polarfire fpga security.” Microsemi, User Guide UG07532, 2018.

[9] Xilinx, “7 series fpga configuration guide,” *UG470 (v1.13.1) August 20, 2018*, 2018.

[10] Intel, “Ug-s10security:intel stratix 10 device security user guide,” 2019.

[11] M. Hoffmann and C. Paar, “Stealthy opaque predicates in hardware-obfuscating constant expressions at negligible overhead,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 277–297, 2018.

[12] K. D. Pham, E. Horta, and D. Koch, “Bitman: A tool and api for fpga bitstream manipulations,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 894–897, IEEE, 2017.

[13] H. M. Kamali, K. Z. Azar, K. Gaj, H. Homayoun, and A. Sasan, “Lut-lock: A novel lut-based logic obfuscation for fpga-bitstream and asic-hardware protection,” in *Proceedings VLSI (ISVLSI) 2018 IEEE Computer Society Annual Symposium on. EH-2001*, pp. 405–410, IEEE, 2018.

[14] M. T. Rahman, K. Xiao, D. Forte, X. Zhang, J. Shi, and M. Tehranipoor, “Ti-trng: Technology independent true random number generator,” in *Proceedings of the 51st Annual Design Automation Conference*, pp. 1–6, ACM, 2014.

[15] A. S. Tahir, “Design and implementation of rsa algorithm using fpga,” *Int. J. of Computers and Technol*, vol. 14, pp. 6361–7, 2015.

[16] https://opencores.org/projects/sha3, “Sha3 keccak design,” 2018.

[17] Speedtest, “Speedtest global index.” ”https://www.speedtest.net/global-index, retrieved,September23,2019”.

[18] R. F. Rizzolo, T. G. Foote, J. M. Crafts, D. A. Grosch, T. O. Leung, D. J. Lund, B. L. Mechtly, B. J. Robbins, T. J. Slegel, M. J. Tremblay, *et al.*, “Ibm system z9 efuse applications and methodology,” *IBM Journal of Research and Development*, vol. 51, no. 1.2, pp. 65–75, 2007.