Secure Boot and Remote Attestation in the Sanctum Processor

Ilia Lebedev
MIT CSAIL, USA
ilebedev@csail.mit.edu

Kyle Hogan
MIT CSAIL, USA
klhogan@csail.mit.edu

Srinivas Devadas
MIT CSAIL, USA
devadas@mit.edu

***Abstract*—During the secure boot process for a trusted execution environment, the processor must provide a chain of certificates to the remote client demonstrating that their secure container was established as specified. This certificate chain is rooted at the hardware manufacturer who is responsible for constructing chips according to the correct specification and provisioning them with key material. We consider a semi-honest manufacturer who is assumed to construct chips correctly, but may attempt to obtain knowledge of client private keys during the process.**

Using the RISC-V Rocket chip architecture as a base, we design, document, and implement an attested execution processor that does not require secure non-volatile memory, nor a private key explicitly assigned by the manufacturer. Instead, the processor derives its cryptographic identity from manufacturing variation measured by a Physical Unclonable Function (PUF). Software executed by a bootloader built into the processor transforms the PUF output into an elliptic curve key pair. The (re)generated private key is used to sign trusted portions of the boot image, and is immediately destroyed. The platform can therefore provide attestations about its state to remote clients. Reliability and security of PUF keys are ensured through the use of a trapdoor computational fuzzy extractor.

We present detailed evaluation results for secure boot and attestation by a client of a Rocket chip implementation on a Xilinx Zynq 7000 FPGA.

I. Introduction

A. Attested Execution

In order for a client to securely execute a remote program it must be able to verify information about the host system. Specifically, clients often desire their computations to be run on trusted hardware, such as Intel SGX or Sanctum [10], [32], [37], that provides guarantees about the privacy and integrity of the computation. For the client to be confident that their remote computation is indeed being run on such a system, it is necessary for the hardware to have a unique and immutable cryptographic identity. A system equipped with such hardware can use this identity to provide the client with an attestation to its state that can be compared against a manufacturer list of authentic platforms. In practice, these attestations can be made anonymous and unlinkable in that an attestation does not reveal who produced it and no two attestations can be determined to have the same origin.

In addition to authenticity, it is important that the processor demonstrate that it has booted correctly, particularly that any code that could affect the attestation to or subsequent execution of client code is unmodified from its expected value. Attested execution thus requires a *secure boot* process in either a local or remote client setting. A bootloader must configure the system as desired by the client and, after booting, the processor needs to prove that the secure container requested by the client has been set up properly, i.e., with appropriate privacy and integrity properties. To accomplish this, the processor presents a container measurement certificate that can be verified by a client.

In one common setting, processor hardware or firmware in a boot ROM cryptographically measures the bootloader software and all software loaded by the bootloader, then uses the processor's secret key to sign the measurement, producing a certificate. The public key associated with the processor's secret key is signed by the manufacturer. The local or remote client verifies the signature of the processor's public key using the manufacturer's public key, verifies the measurement certificate using the processor's public key, and checks the measurement itself against a client-side stored hash of the expected system state.

To produce such a certificate the processor must be provisioned with a public key pair. Generally, the manufacturer produces a key pair for each chip and embeds the private key into the chip, typically into secure non-volatile memory. In this scenario however, the manufacturer knows the chip's private key without the chip leaking it so even a correctly manufactured processor could have exposed keys at production. Alternatively, to maintain privacy of the secret key, a hardware True Random Number Generator (TRNG) generates a random seed stored in secure non-volatile memory, which is used to generate a public/private key pair inside the processor. The manufacturer signs the public key, which can be stored in publicly readable non-volatile memory while the private key never leaves the processor and is unknown to the manufacturer. Thus, while the manufacturer must be *honest*, we tolerate a "curious" manufacturer, as they do have to be trusted to manage secrets.

An alternative scenario involves the use of silicon Physical Unclonable Functions (PUFs) [17]. PUFs extract volatile secret keys from semiconductor manufacturing variation that only exist when the chip is powered. The first documented use of PUF generated keys in a secure processor setting was in the Aegis processor [41]. The PUF was used to generate a symmetric key shared with the client through a cryptographic protocol. PUFs are used as symmetric key generators in

commercial products such as Xilinx Ultrascale Zynq FPGAs [48][1] and Intel/Altera FPGAs [2]. While it has been well known that a PUF can be used to generate a random seed for a public/private key generator inside of a secure processor (e.g., [12]), we are unaware of any published implementation that accomplishes this.

B. Our approach

Using the RISC-V Rocket chip architecture [27] as a base, we design, document, and implement an attested execution processor that does not require a private key explicitly assigned by the manufacturer. Instead, the processor derives its cryptographic identity from manufacturing variation measured by a PUF. Software executed by a bootloader built into the processor transforms the PUF output into an elliptic curve key pair, the public portion of which is signed by the manufacturer using its private key. Since a regenerated PUF output needs to be error corrected, we employ a trapdoor computational fuzzy extractor [20] to ensure the security of error correction. The (re)generated private key is used to sign trusted portions of the boot image, and is immediately destroyed. The signed boot image is responsible to performing attestations to remote clients. A client can establish a secure communication channel with the platform using Diffie-Hellman key exchange and request the platform signature on the boot image. It can verify this signature to establish authenticity and integrity of the system before sending its code and data. Upon receiving the client code, the boot image can further attest to the state of the client program prior to execution. We argue that such a processor is attractive when physical security is important, and when it is desirable to keep the handling of secret information by entities to a minimum.

We present detailed evaluation results for key generation, secure boot and attestation for a Rocket chip implementation on a Xilinx Zynq 7000 FPGA.

C. Organization of this paper

The rest of this manuscript is organized as follows: Section II describes prior work and introduces relevant concepts. Section III defines the threat model used throughout this manuscript. Section IV describes key derivation performed at boot and the hardware the boot ROM root of trust assumes for correct operation. Our remote attestation protocol is described in Section V. Section VI describes primitives to implement the root of trust key derivation. Section VII explores the tradeoffs and performance of the root of trust; Section VIII concludes.

II. Background and Related Work

A. Secure Boot

In order to establish a trusted environment for program execution, the host system must first have booted into a verifiable state. If a step in the boot process is not included in the attestation to the client then its state cannot be guaranteed and it could potentially compromise the privacy or integrity of any subsequently loaded programs. A trusted bootloader is typically the first component in this process and is considered the root of trust for the boot process. It is the first code to run upon system initialization and is responsible for checking the measurements and/or signatures of subsequent components either locally or by utilizing a piece of trusted hardware such as a TPM.

The primary distinctions between different secure boot processes used in practice are how they obtain their attestation root key, whether the root of trust for measurement/verification differs from the root of trust for attestation, and whether components are verified using a signature from the manufacturer or by a measurement of their code.

Heads [24] implements a measured boot process supported by a TPM where the root of trust for the boot process is a write protected ROM and the root of trust for authentication and attestation is the TPM. ARM's TrustZone does not provide a canonical mechanism for remote attestation, but software in its secure world is able to implement its own attestation. Intel's Secure Boot has both measured and verified modes. For both modes microcode on the CPU is the root of trust for the boot process [35]. In the measured mode a TPM is responsible for storing and attesting to the measurements while in verified mode each component is signed by the manufacturer and these signatures are verified prior to loading the component. In contrast, SGX is not concerned with the boot process of the system in general and focuses only on providing attestation and authentication for enclaves post boot using keys stored on the processor [4], [32].

In all three cases, the TPM manufacturer and potentially the CPU manufacturer are directly responsible for provisioning platform attestation/authentication keys.

B. Trusted key derivation

Commercial secure processors such as Intel Skylake (which includes SGX) and ARM's secure processor offerings (based on TrustZone) appear to use some form of non-volatile memory to store a secret key. SGX patents disclose the use of a PUF to generate a device-specific key from a fused key obfuscated in the design [9].

The Aegis processor was introduced in [39], and [41] described an implementation that used a PUF to generate a symmetric key for use in attestation. The error correction scheme used predated fuzzy extractors and was fairly limited in its capability [16].

Prior work in PUF based key derivation is described below.

1) Physically Obfuscated Keys and Fuzzy Extractors: Silicon PUFs were introduced in [17], and over the past several years there have been several proposals for candidate silicon PUF architectures, which include delay PUFs [40] and SRAM PUFs [23], and several variants. Our focus here is on error correction of PUF outputs, so the error-corrected output can be used in cryptographic applications. A PUF is being used as a Physically Obfuscated Key (POK) [16]. Delay PUFs naturally provide

[1] The PUF generated key is used to AES-encrypt and decrypt keys that encrypt and decrypt the FPGA bitstream. Not even the manufacturer knows the PUF key.

"confidence" information, which provides additional knowledge of the stability of a POK bit [40].

Silicon POK key generation was first introduced using Hamming codes in [16] and more details were presented in [38]. The security argument used is information-theoretic. Specifically, if one requires a k-bit secret from n bits generated by the POK, then at most $n - k$ bits could be exposed. The number of correctable errors is quite limited in this approach.

Fuzzy extractors [13] convert noisy biometric data (either human or silicon) into reproducible uniform random strings, which can then serve as secret keys in cryptographic applications. In the fuzzy extractor framework, it is possible to extract near-full-entropy keys from a POK source while maintaining information-theoretic security. The information-theoretic security, however, comes at a high cost in terms of the raw entropy required and the maximum tolerable error rate. Even in cases where entropy remains after error correction (e.g., [31]), there may not be enough entropy remaining to accumulate the necessary bits for a 128-bit key.

There are several works that created helper data that is information-theoretically secure. A soft-decision POK error correction decoder based on code-offset was described in [29], [30] where the confidence information part of the helper data was proven to be information-theoretically secure under an i.i.d. assumption.

[50] uses POK error correction helper data called Index-Based Syndrome (IBS), as an alternative to Dodis' code-offset helper data. IBS is information-theoretically secure, under the assumption that POK output bits are independent and identically distributed (i.i.d.). Given this i.i.d. assumption, IBS can expose more helper data bits than a standard code-offset fuzzy extractor construction. Efficiency improvements to IBS that maintained information-theoretic security are described in [21] and [22].

2) Computational Fuzzy Extractors: Fuller et al. [15] give a computational fuzzy extractor based on the Learning With Errors (LWE) problem. In Fuller et al.'s scheme, the output entropy improves; the error correction capacity, however, does not. Indeed, Fuller et al. show in their model that computational fuzzy extractors are subject to the same error correction bounds as information-theoretic extractors. Their construction therefore requires exponential time to correct $\Theta(m)$ errors, where m is the number of bits output by the POK.

Fuller et al. expect that the exponential complexity in correcting a linear number of errors is unlikely to be overcome, since there is no place to securely put a trapdoor in a fuzzy extractor. Herder et al. [20] recognized that certain kinds of silicon biometric sources have dynamically regenerated confidence information that does not require persistent storage memory and can in fact serve as a trapdoor. This results in an efficient extractor in terms of the amount of helper data and the entropy requirement on the POK, though the computational requirements of error correction may be higher than in information-theoretic schemes. However, in the secure processor application of this paper, error correction can be done in software making Herder et al.'s scheme attractive; the additional hardware that is required is a set of ring oscillators corresponding to the POK.

C. Remote attestation

Attestation by hardware components of a platform allows remote clients to verify details of the system state. In particular, clients need to verify both the authenticity of the system performing the attestation and the integrity of their own remote application. Platform authenticity is typically verified via its membership in a manufacturer managed group of all non-revoked platforms. During the manufacturing process platforms are provisioned with public key pairs where the secret key is permanently bound to the platform. Group membership can be determined by verifying signatures produced by the platform as only platforms with valid secret keys are able to generate signatures that can be verified by one of the public keys for the group. These schemes can also be anonymous and unlinkable where the verifier learns only the platform's membership in a specified group, but not which platform specifically produced the signature or whether any pair of signatures was produced by the same platform [7].

These platforms can then be responsible for signing the hash of a program's state to demonstrate to a client that it was loaded as expected. The client's trust in this process is rooted in the provisioning of the platform's secret key. Often, keys are generated and provisioned explicitly by the manufacturer, but it is also possible to generate keys on platform using a TRNG or PUF in such a way that the manufacturer does not acquire knowledge of the platform's secret key.

Intel SGX, TXT, and TPMs are all able to provide attestations to remote clients about platform state using keys directly provisioned by the device manufacturer. TPM's use Direct Anonymous Attestations (DAA) where the signatures do not reveal the identity of the signer unless the secret key used to generate the signature is subsequently revealed [1].

D. Isolated execution

XOM [28] introduced the idea of executing sensitive code on data in isolated containers with an untrusted operating system handling resource allocation for the secure container.

Aegis [39] instead relies on a trusted security kernel, the hash of which is collected at runtime and used in conjunction with the hash of secure containers to derive container keys and attest to the initial state of the container. Unlike XOM, Aegis tolerates untrusted memory.

Intel's Trusted Execution Technology (TXT) [19] is a widely deployed implementation for trusted execution. However, it was shown to be susceptible to several confused deputy attacks [46], [47] that allowed a malicious operating system to direct a network card to perform direct memory accesses on the supposedly protected virtual machine. A later revision to TXT introduced DRAM controller modifications that selectively block DMA transfers to mitigate these attacks.

Intel's SGX [4], [32] adapted the ideas in Aegis and XOM to multi-core processors with a shared, coherent last-level cache. It is widely deployed in Skylake and later generation

processors and supports isolated execution and attestation of client processes in the presence of a malicious operating system. It remains vulnerable to side channel and controlled channel attacks [6], [49].

Sanctum [10] offers the same promise as Intel's Software Guard Extensions (SGX), namely strong provable isolation of software modules running concurrently and sharing resources, but protects against an important class of additional software attacks that infer private information from a program's memory access patterns. Sanctum follows a principled approach to eliminating entire attack surfaces through isolation, rather than plugging attack-specific privacy leaks. Most of Sanctum's logic is implemented in trusted software, called the "Security Monitor", which does not perform cryptographic operations using keys. The Sanctum prototype targets a Rocket RISC-V core, an open implementation that allows any researcher to reason about its security properties. The Sanctum prototype does not implement memory encryption implying that DRAM is trusted.

ARM's TrustZone [3] is a collection of hardware modules that can be used to conceptually partition a system's resources between a secure world, which hosts a secure container, and a "normal" world, which runs an untrusted software stack. The TrustZone documentation [5] describes semiconductor intellectual property cores (IP blocks) and ways in which they can be combined to achieve certain security properties, reflecting the fact that ARM is an IP core provider, not a processor manufacturer. Therefore, the mere presence of TrustZone IP blocks in a system is not sufficient to determine whether the system is secure under a specific threat model. The reset circuitry in a TrustZone processor places it in secure mode, and executes a trusted first-stage in an on-chip ROM. TrustZone's TCB includes this bootloader in addition to the processor and all peripherals that enforce access control for the secure world.

III. Threat Model: An honest but curious manufacturer

Our goal is to use a PUF in conjunction with a root of trust endorsed by the manufacturer to improve the security of key derivation over the case where the manufacturer is responsible for explicitly provisioning keys. In this setting the manufacturer is no longer able to store keys and thus cannot provision multiple systems with the same key or retroactively compromise keys.

We consider an "honest" manufacturer with a well-known public key that follows the specification for constructing the PUF and CPU without introducing backdoors. The manufacturer is also responsible for endorsing the correct root of trust and not replacing it with a malicious program that does not follow the protocol in Section IV. The manufacturer is "curious", however, and may attempt to learn information while following the specification, including repeating a protocol multiple times with different parameters. As a concrete example, we handle a manufacturer that attempts to learn information about a PUF output through many runs of the key derivation protocol.

The root of trust (boot ROM) is assumed to correctly derive keys from the PUF output, and then measure and endorse a "payload" software. Much like the hardware itself, we assume this implementation to be honest and free of bugs, as it is not replaceable on a given chip.

We assume the system memory is trusted and only accessible via the processor, which is able to sanitize this memory at boot, and guard access to cryptographic keys and other private information stored there. If DRAM must be excluded from the TCB, one may employ transparent encryption at the DRAM controller (counter mode AES on the data of all or some memory accesses) as in XOM [28] and Aegis [39]. Moreover, if the threat model must protect the access pattern of DRAM requests – as in the case of a hostile machine operator observing traffic on the DRAM link – an Oblivious RAM [18] primitive may be employed, with corresponding overhead as in the Ascend processor [14].

The Sanctum processor [10] implements a strongly isolated software container (enclave) primitive, which can be used to dramatically reduce the size of a system's software trusted computing base. Sanctum relies on a trusted (but authenticated) "Security Monitor" (precisely the payload in this secure boot scheme), which correctly configures the underlying hardware, and maintains the invariants needed for the system's security argument. Sanctum excludes the operating system and any software it runs from the TCB (these are therefore loaded *after* the secure boot, and are not measured), but allows the creation and remote attestation of a secure container, thereby bootstrapping trust in additional software (whereby the security monitor uses its keys to measure and sign the secure container). While Sanctum's security monitor is trusted to correctly implement its API, it is authenticated, meaning the remote party seeking attestation is able to reject attestations coming from a system that loaded a known-vulnerable security monitor.

IV. Root of Trust

At reset, the processor must execute instructions beginning at an address corresponding to a trusted "boot ROM" – a requirement described in Section VI-A. This first-stage bootloader, which we denote the "root of trust" is a program that loads a "payload" binary segment from untrusted storage into trusted system memory, derives the payload-specific keys (SK_P, PK_P), and uses the device key to sign PK_P and the trusted manufacturer's endorsement of the processor.

Several implementation variations on this common theme are explored in this section, differing in their construction of SK_{DEV} and their use of the trusted boot ROM memory. Figure 1 shows a general template for all roots of trust examined in this manuscript.

The processor has multiple cores, all of which execute starting at the same address after a reset. Our root of trust stalls all but one core while executing the root of trust program on `Core 0`. Because the contents of memory at

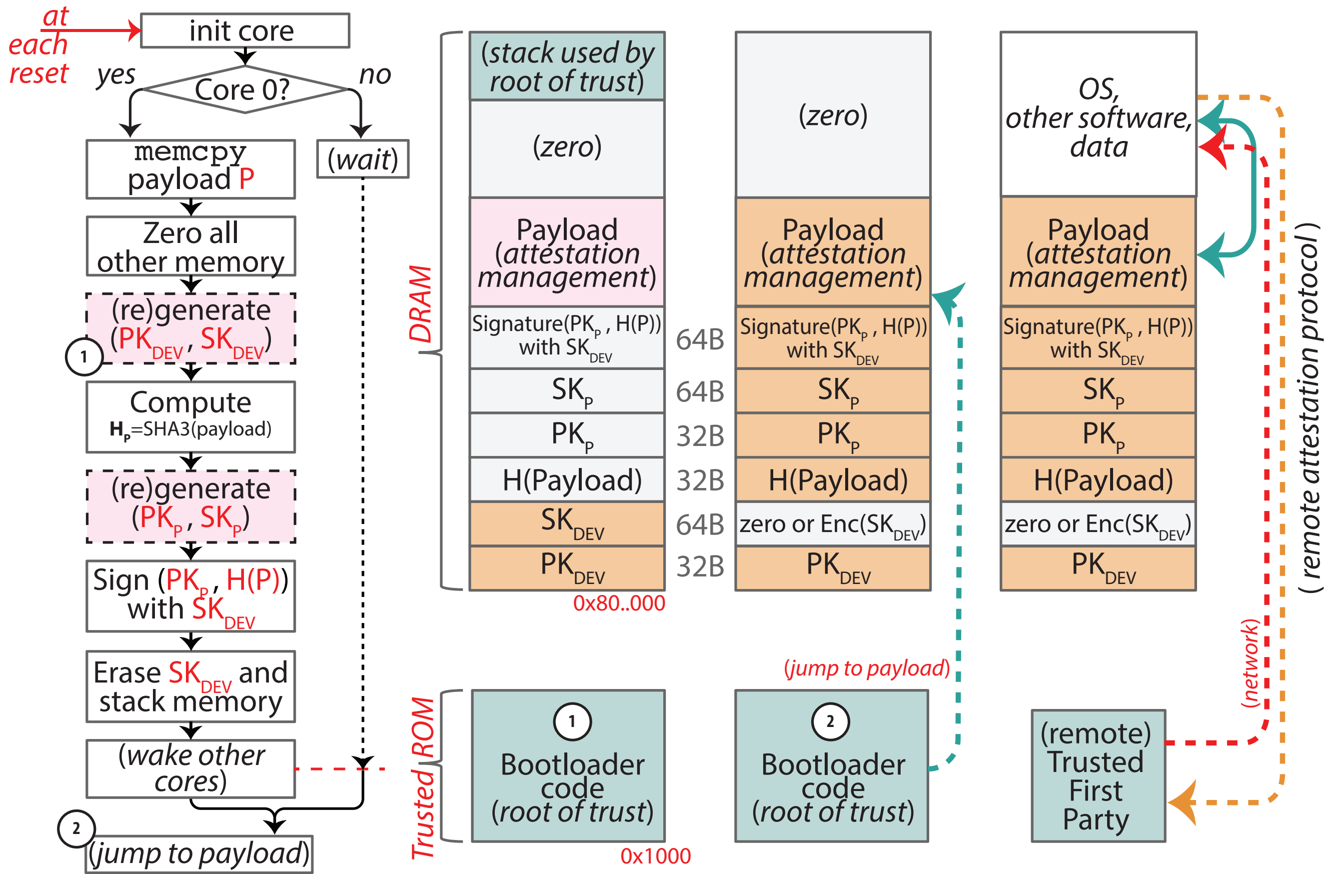

Fig. 1. The root of trust signs trusted system software (Payload) and its derived key pair (PK_P, SK_P) at boot. During remote attestation, the trusted first party communicates with the processor (or an enclave, in the case of Sanctum), creates an "attestation", and establishes a shared key with the first party.

boot are undefined, a synchronization via a lock in memory is not possible; instead, this work relies on all but one core waiting on an interrupt. `Core 0` resumes all other cores via an inter-processor interrupt at the completion of the root of trust program. All cores sanitize core-local state (registers, configurations, etc.) in order to prevent an adversary from leaking private information via soft reset into a malicious payload. For the same reason, the root of trust must erase all uninitialized memory via a `memset`.

In implementing the root of trust, we rely on a portable implementation of the SHA3 hash function [36] and ed25519 elliptic curve DSA cryptography (we slightly modify an existing implementation [34] to use SHA3). For the roots of trust that use an AES cipher, we again rely on an existing, portable implementation [26]. The sections below describe our root of trust implementations that derive cryptographic keys for the processor and a given payload.

A. Deriving an ephemeral SK_{DEV} for cloud-hosted FPGA designs

Here, we describe a root of trust in a setting where an honest but curious party (the "manufacturer") has access to the processor at boot, and is available to sign the processor's randomly generated public key (SK_{DEV}) messages with their

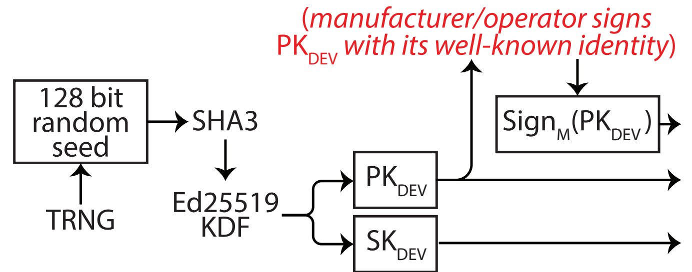

Fig. 2. Ephemeral SK_{DEV} derivation from entropy.

well-known key. In other words, the "manufacturer" is able to locally attest the processor, obviating the need for a persistent cryptographic root in the device. A practical example of this setting is an FPGA secure processor deployed by a semi-trusted cloud provider. The hardware platform hosting the secure processor is time-shared among many mutually distrusting tenants, and the provider does not guarantee the processor is deployed on a specific FPGA. In this scenario, the cloud operator is effectively a "manufacturer" of an ephemeral secure processor; non-volatile keys are neither meaningful nor easy to implement in this scenario.

The ephemeral SK_{DEV} key derivation is illustrated in Figure 2. Here, device keys (PK_D, SK_D) are derived from a hashed random value; each instance of the secure processor has a new cryptographic identity, which does not survive a reset. The processor commits to (PK_D) at boot, and expects the manufacturer (machine that programs the cloud FPGA) to endorse it with its well-known cryptographic key $(Sign_M(PK_{DEV}))$. This signature binds the processor's PK_{DEV} to the remote user's trust in the manufacturer's integrity: the processor eventually performs attestation involving a certificate signed with SK_{DEV} (cf. Section V), and presents $Sign_M(PK_{DEV})$ to convince the remote user that the certificate is rooted in a trustworthy platform, not an arbitrary malicious emulation of the remote attestation protocol.

B. Non-volatile SK_P derived from a PUF

In a situation where no trusted party can locally attest to the processor at boot, e.g., a computer system deployed remotely in an untrusted environment, remote attestation *must* be rooted in a trusted public key. This key must persist across reboots, and must not be practically recoverable by an adversary.

While the device *could* be provisioned with a key pair explicitly, via a tamper-resistant non-volatile memory, this would assume a lot of trust in the party installing the key. A "curious" manufacturer may choose to remember the keys, compromising a processor retroactively by leaking its secret key, or endowing an untrusted emulator with the same key pair.

To satisfy the threat model presented earlier (cf. Section III), we employ a root of trust that repeatedly generates its cryptographic key pair by seeding a Key Derivation Function (KDF) with a physically obfuscated key via a fuzzy extractor scheme proposed by Herder et al. [20]. The roots of trust, **P256** and **P512** use an array of identical ring oscillator pairs (a physically obfuscated key, where manufacturing variation informs the relative frequency of the oscillators in each pair) and a trapdoor fuzzy extractor to derive a semi-repeatable secret M-bit $\vec{e}$ (a M-element vector in $GF(2)$), where M is 256 and 512 for **P256** and **P512**, respectively (this is the only difference between the two implementations). Section VI-B details this hardware structure and its parameters.

1) Initial key provisioning by the manufacturer with the PUF: After the secure processor is manufactured (or the FPGA is programmed with the design containing the secure processor), the root of trust *provisions* a secret 128-bit $\vec{s}$. From $\vec{s}$, the root of trust computes a public M-bit vector $\vec{b} = A\vec{s} + \vec{e}$, where A is an M-by-128 matrix in $GF(2)$. $\vec{b}$ and A are public "helper data", which are used by future invocations of the root of trust to recover $\vec{s}$.

The key provisioning is illustrated in Figure 3. From $\vec{s}$ (hashed), an ed25519 key derivation function computes (PK_{DEV}, SK_{DEV}). The root of trust program then runs to completion, as shown in Figure 1, conveying $(PK_{DEV}, \vec{b}, ...)$ (but not $\vec{s}$ or SK_{DEV}, which are erased by the root of trust) to the payload. The manufacturer is able to learn PK_{DEV} via the payload, and sign this public key with their own well-known cryptographic identity.

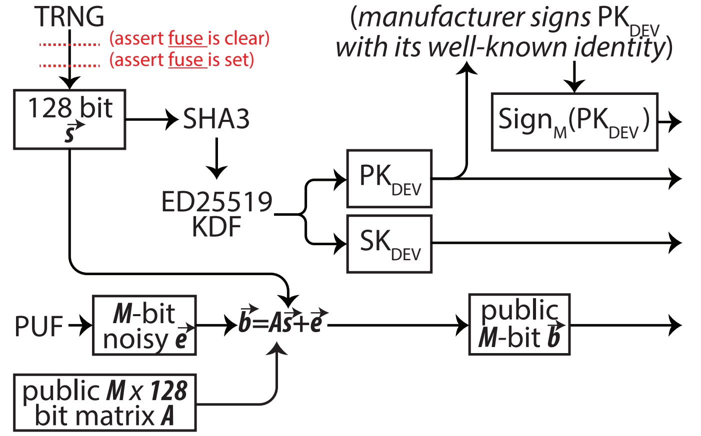

Fig. 3. Initial provisioning of PUF-backed device keys.

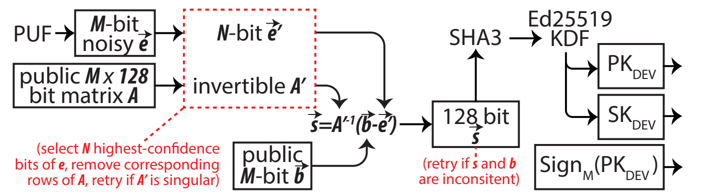

Fig. 4. Key derivation of PUF-backed device keys.

P256 and **P512** employ the TRNG to source a 128-bit random value for $\vec{s}$, and use the same A (256-by-128 and 512-by-128 bit matrices, for **P256** and **P512**, respectively) for all devices, by including A as part of the boot ROM. The manufacturer selects the matrix A by generating a string of random bits of the requisite length.

To prevent the manufacturer from repeatedly re-provisioning keys (which may weaken the secrecy of $\vec{e}$), the root of trust requires a fuse (one-time non-volatile memory) to be written as part of provisioning. Due to a quirk of the FPGA platform, the root of trust program does not write the fuse itself. Instead, before selecting $\vec{s}$, the program ensures a fuse (exposed as a readable register) is *not* set, and stalls until its value changes (a fuse cannot be cleared after being set, so provisioning proceeds exactly once). The manufacturer sets the fuse via a management interface (JTAG). If the fuse is set initially, the root of trust does not provision a new $\vec{s}$, and instead attempts to recover it from helper data in untrusted memory, as described below.

We note that the fuse can be avoided by making a stronger assumption about PUF bits or a stronger assumption on the hardness of a variant Learning Parity with Noise (LPN) problem [20].

2) Key recovery with the LPN PUF: At each reset, the root of trust *reconstructs* 128-bit secret $\vec{s}$ from public $(\vec{b}, A)$, and uses it to re-compute (PK_{DEV}, PK_{DEV}) as illustrated in Figure 4. The root of trust reads the array of ring oscillator pairs to obtain $\vec{e}$ (a somewhat noisy view of the bit vector

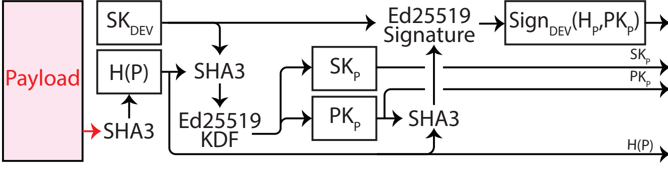

Fig. 5. Key derivation and endorsement by the device of the Payload.

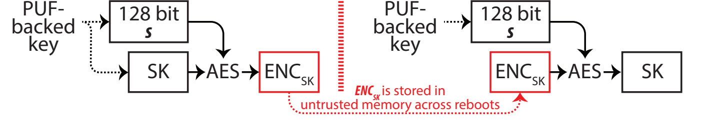

Fig. 6. Key encryption for untrusted storage.

used in provisioning $\vec{s}$), and the corresponding "confidence" information $\vec{c} \in \{\mathbf{Z}\}^M$. As described in the original construction [20], the root of trust discards low-confidence bits of $\vec{e}$ and corresponding rows of $\vec{b}$ and A, resulting in $\vec{e}'$, $\vec{b}'$ and A', respectively. The bits of $\vec{e}$ with high confidence values are assumed to be stable across reboots with high probability, a design requirement for the ring oscillator pairs, as described in Section VI-B and evaluated in Section VII-C.

If A' is invertible, the root of trust uses Gauss-Jordan elimination (in $GF(2)$, informed by [25]) to compute A'^{-1}. A different A' is chosen if A' is singular. A' is a sampling of 128 128-bit rows from a larger random matrix, so we consider A' to be a random 128-by-128 matrix in $GF(2)$. The probability [43] of A' having a zero determinant (and therefore singular) is given by $P(n=128, q=2) = 1-(1-q^{-1})(1-q^{-2})...(1-q^{-n}) = 1 - \prod_{i=1}^{128} 1 - \frac{1}{2^i} \approx 71.12\%$. The probability that A' is invertible is therefore 28.88%, and the root of trust considers ~ 3.5 matrices before finding one that is invertible, in expectation. Given an invertible A', the root of trust straightforwardly computes $\vec{s}' = A'^{-1}(\vec{b}' - \vec{e}')$, and verifies $\vec{s}' = \vec{s}$ by computing $\vec{b}'' = A\vec{s}' + \vec{e}$; a very high edit distance between b and b'' signals an uncorrected error, and retries with a new $\vec{e}$. Section VII-C examines the (very low) expectation of bit errors among high-confidence elements of $\vec{e}$.

After s is recovered, the root of trust computes (PK_{DEV}, SK_{DEV}) from a KDF over its hash, as before. The root of trust *disables* the PUF ring oscillator readout until the next reset, as described in Section VI-B in order to prevent a malicious payload from accessing the secret $\vec{e}$.

C. Payload keys and endorsement

All roots of trust considered in this manuscript identically derive the payload-specific keys (PK_P, SK_P) from a SHA3 hash of the payload and SK_{DEV}. The key derivation is shown in Figure 5; all software libraries needed to do this were previously introduced to derive device keys. The root of trust computes $H(P) = SHA3(Payload)$, combines it with SK_{DEV}, and uses the result to seed an ed25519 KDF. $(PK_P, SK_P) = KDF_{ed25519}(SHA3(SK_{DEV}, H(P)))$.

The payload is *not* assumed to be honest; a malicious payload may leak its own hash and keys. This does not compromise the processor's key SK_{DEV}: the SHA3 hash is a one-way, collision-resistant function, so even should an adversary reconstruct the seed from which its keys were derived, and although $H(P)$ is public, the other hash input (SK_{DEV}) is not compromised. SK_P, meanwhile, is derived, in part, from the payload's hash, so an adversary leaking their *own* keys has no bearing on the confidentiality of an honest payload's keys.

Finally, the processor "endorses" the payload by signing its hash and public key: $Certificate_P = Signed_{SK_{DEV}}(SHA3(H(P), PK_P))$. Given the certificates $(Certificate_{DEV}, Certificate_P)$ and SK_P, an uncompromised payload can prove to a remote party that it is indeed a specific piece of software executing on hardware signed by the trusted manufacturer, as we will describe in Section V. The payload is expected to safeguard its key (any payload that leaks its SK_P can be *emulated* by an adversary, and is not trustworthy).

All intermediate values are, as before, on a stack in memory, which must be erased before the processor boots the untrusted payload.

D. Key encryption and minimal root of trust

In the case of the $\mathbf{P256_{AES}}$ and $\mathbf{P512_{AES}}$ roots of trust, the processor encrypts its secret keys $(SK_D, SK_{Payload})$ with a symmetric key derived from the PUF, to be kept in untrusted non-volatile memory, as shown in Figure 6. To this end, the root of trust links a portable implementation of the AES in addition to the SHA3 and ed25519 KDF, and includes control flow to perform the ed25519 KDF, which will be used when no encrypted keys are provided with the payload. The payload is responsible for storing its own encrypted key blob, as this value is public and untrusted. As shown in Section VII, decrypting a stored key with AES does offer a performance advantage over re-deriving the key using the KDF at each boot. Persistently maintaining the key to be decrypted with AES requires additional complexity, however, and a larger boot ROM to accommodate the AES code. This increases the trusted code base, so implementations should consider this tradeoff when designing the root of trust.

We consider also a root of trust intended to minimize the size of the boot ROM by securely loading the previously described root of trust from untrusted memory (We denote these five roots of trust $\mathbf{HT}$, $\mathbf{HP256}$, $\mathbf{HP256_{AES}}$, $\mathbf{HP512}$, and $\mathbf{HP512_{AES}}$). Here, the trusted boot ROM only includes the instructions needed to copy a root of trust binary from *untrusted* memory to DRAM, and verify its hash against an expected constant. The boot ROM consists of `memcpy`, `SHA3`, a literal expected hash, and a software loop to perform the hash comparison. Aside from a variation in the size of the root of trust, and different expected hash constants, the five designs produce similar binaries, of nearly identical size, as shown in Section VII.

V. Remote Attestation

An immutable hardware root of trust as described in the previous section allows the platform to provide attestations about its state to remote clients. Use of a PUF to generate platform root keys implies that the platform's keys are both unique to the system running client code and unknown to any other party. This gives remote clients high confidence in the authenticity of any attestations provided by the platform.

A. Remote Attestation with Generic Payload

The payload loaded as part of the system's boot process is responsible for handling remote attestation of client processes. The initial state of the payload is verified as part of the key generation process, but it must be trusted to maintain its integrity and the privacy of its keys, PK_P and SK_P, that were derived by the device during boot.

The general procedure the payload follows to perform attestations is outlined in Figure 1 from the initial generation of keys during the boot process through the sending of the attestation to the client. The remote client must first initiate a Diffie-Hellman key exchange to establish a secure communication channel between itself and the platform. Upon receiving the Diffie-Hellman parameters $g,\ g^A,\ p$ from the client, the payload will then choose a B and compute g^B. It will then send a signature under its secret key, SK_P, of the public key of the device (PK_{DEV}), the signature under the device secret key of the payload's public key and hash of initial payload state ($Sign_{SK_{DEV}}(PK_{payload}, H(payload))$), and the Diffie-Hellman parameters back to the client. The signature from the device containing the payload's public key and a hash of its initial state can be used by the client to verify that the payload loaded by the device matches the one it expected. The client can then use the secure communication channel that has been established to send over any code and data that it wishes to be run by the platform and will receive a signature from the payload over a hash of the state of the client's program. In this way the client can bootstrap trust in their remote code from their trust in the secure processor.

B. Remote Attestation with Sanctum

A more detailed protocol as performed by Sanctum is shown in Figure 7. The primary distinction between the two protocols is that the payload in the case of Sanctum is its security monitor which is responsible for enforcing some of the isolation properties for client enclaves as well as spawning a special "signing enclave". The security monitor hashes a client enclave as it is initialized, and delegates its endorsement to the signing enclave, which is exclusively able to access SK_P. We rely on the isolation properties of Sanctum enclaves to guarantee privacy of the signature.

To avoid performing cryptographic operations in the security monitor, Sanctum instead implements a message passing primitive, whereby an enclave can receive a private message directly from another enclave, along with the sender's measurement. Details are provided in [11].

Sanctum is also capable of handling local attestations without the presence of a trusted remote party. In this use case requests for attestation are authenticated by the signing enclave in order to prevent arbitrary enclaves from obtaining another enclave's attestation.

C. Anonymous Attestation

Neither of the protocols outlined here provide anonymous attestations, but the construction is not incompatible with anonymous schemes as the attestation protocol is implemented in replaceable authenticated software. In order to convert the existing scheme to an anonymous one a trusted party is needed to handle group membership and revocation. This party, often the platform manufacturer, is responsible for managing the set of public keys corresponding to authentic devices. It must also keep track of any compromised platforms and add their keys to a revocation list.

As an example, to implement direct anonymous attestation a platform wishing to prove that it is authentic will demonstrate to the manufacturer in zero knowledge that it possesses a secret key corresponding to one of the public keys on the list. This can be done using a scheme for signatures over encrypted messages such as [7], [8] and allows the manufacturer to certify that the platform is authentic without learning which platform corresponds to a key and breaking its anonymity. To perform attestations the platform will generate a signature proof of knowledge over the value to be attested to, the certification from the manufacturer, and its secret key. Remote clients can verify based on the platform's knowledge of a certification from the manufacturer over its secret key that the attestation came from an authentic device. These attestations are guaranteed to be both anonymous in that they reveal no information about which platform generated them and unlinkable in that no two attestations can be determined to have come from the same platform.

For the use case of allowing remote clients to verify that their outsourced computation is being run on the expected hardware, anonymity does not provide any immediate benefit, as the same entity, a cloud provider for example, will often own a large set of interchangeable platforms on which the client could have been scheduled. Anonymity could even be detrimental in the case where the client wishes to verify that the provider scheduled their computation to a specific datacenter and did not further outsource it.

VI. Required Hardware

While the secure boot and remote attestation mechanisms presented in this manuscript are not limited to one specific processor architecture, we do rely on some basic properties and primitives to implement the root of trust key derivation.

Specifically, the processor must boot from a trusted code sequence with an integrity guarantee, e.g., an on-chip ROM, so that the measurement and key derivation procedures described in this work produce deterministic results. An uncompromised system should be able to reliably generate its root keys, while

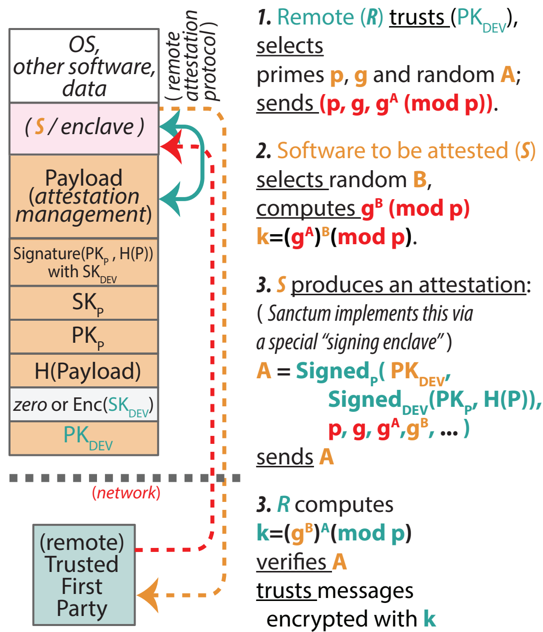

Fig. 7. A protocol diagram for remote attestation, as implemented by the Sanctum processor.

changes to the boot process should result in the system deriving an incorrect key.

Post boot the processor is responsible for maintaining the integrity of the system and the confidentiality of the keys it receives from the root of trust. As keys are generated in part based on the authenticity of the software stack, a machine that boots with malicious or otherwise modified software will generate different keys than the system would have had it booted with the expected software stack. Thus, a malicious software stack may leak *its* keys without compromising the security of an honest software system, as the keys for modified system have no relation to those for the uncompromised software stack. The exact isolation mechanism required to maintain these confidentiality and integrity guarantees depends on the threat model. Sanctum capably achieves these requirements for a software adversary via its enclaves and security monitor, while a system on which *all* software is trusted may rely on process isolation alone.

In addition to these basic requirements, the processor must provide a trusted source of entropy for attestation. In Section VI-C, we describe the implementation of a true random number generator (TRNG), which can be read directly by unprivileged software. In a system where the operating system is part of the trusted code base, OS-provided entropy may suffice.

The processor must also store or derive its root cryptographic key pair (or a seed) in a trustworthy manner. In this manuscript, we focus on PUF-backed keys and keys derived from TRNG entropy, as described in Section VI-B. Other systems may employ non-volatile memory with explicitly provisioned keys, taking care to adjust the threat model to include the additional

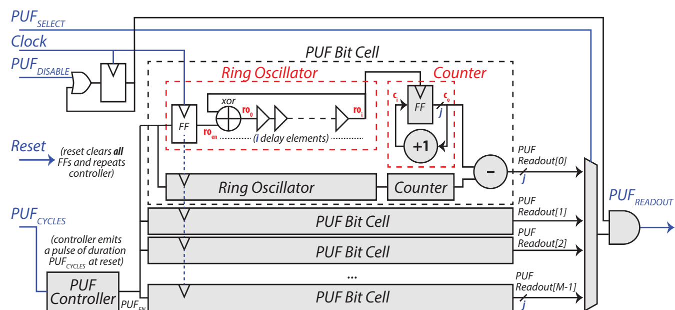

Fig. 8. A detailed block diagram of the LPN PUF.

trust in the manufacturer.

A. Baseline Processor Architecture (Sanctum)

The Sanctum processor [10] is a modification of the Rocket Chip [27] implementation of the RISC-V [45] priv-1.10 [44] instruction set architecture. Rocket Chip's straightforward architecture guarantees that a trusted "boot ROM" executes at each reset.

Sanctum's hardware modifications enable strong isolation (integrity and confidentiality) of software containers (enclaves) with an insidious threat model of a remote software adversary able to subvert system software and actively tamper with network traffic. Sanctum also guarantees the integrity of an honest "Security Monitor" (SM), which is authenticated at boot, and maintains meaningful isolation guarantees, as was formally verified [37]. Sanctum uses RISC-V's "machine mode" (highest privilege software) to host the SM, and maintains that the SM has exclusive access to a portion of DRAM (thus maintaining its integrity and protecting its keys). The SM routes interrupts and configures hardware invariants to ensure enclaves do not involuntarily share resources with any other software. Enclaves access the TRNG via a user-mode CSR (register), meaning enclaved software accesses trusted entropy without notifying the untrusted operating system. Sanctum also guards against subtle side channel attacks via the cache state and shared page tables.

Sanctum trusts the integrity and confidentiality of DRAM, and maintains its security invariants on all DRAM accesses including DMA and the debugger interface. As a consequence, the root of trust must include code to *sanitize all DRAM* before any untrusted software is allowed to execute, in order to guarantee keys persist across reboots and be compromised by malicious system software.

The work described in this manuscript does not rely on details of any specific ISA, and can be adopted for arbitrary processor systems so long as the underlying architecture can provide the basic requirements outlined at the head of this section.

B. LPN PUF for secure, repeatable key derivation

In order to achieve secure non-volatile keys with an honest-but-curious manufacturer, we rely on a trapdoor LPN PUF

scheme constructed by Herder et al. [20]. The majority of the mechanism is implemented in the root of trust software, as detailed in Section IV-B; the required hardware is an array of M identical ring oscillator pairs, as shown in Figure 8. Software accesses this structure via CSRs (control/status registers) at `Core 0`: `puf_disable`, `puf_select`, and `puf_cycles`, and `puf_readout`. Their semantics are explained below.

Each ring oscillator pair is endowed with *counters* to compute the relative frequency of the pair of ring oscillator: the j-bit counters are clocked by each ring oscillator, their counts subtracted to derive a j-bit $magnitude$. The elements $\vec{e}$ and $\vec{c}$ vectors, as defined in Section IV-B2 are the sign and absolute value of each oscillator pair's $magnitude$, respectively. The oscillator (and counter) circuits are identically implemented to minimize bias due to differences in design among the ring oscillators – this circuit measures bias from manufacturing variation. The ring oscillator is a series of delay elements (buffers), feeding back into an inverting element (XOR) at the head. Each ring oscillator consists of i delay elements in total. A controller circuit generates a pulse of `puf_cycles` processor clock cycles at reset, simultaneously enabling all ring oscillators. While enabled, the ring oscillates, and the resulting signal is used to clock the attached counter circuit. The specific parameterization of i and j depends on the implementation platform: the ring oscillators must not oscillate at a higher frequency than the counter circuit can accommodate. Depending on the magnitude of manufacturing variation, j can be adjusted to increase the dynamic range of counters. A more sophisticated, future design may implement a saturating counter to better address a platform where high dynamic range of counts is expected.

The processor can adjust `puf_cycles` and trigger a soft reset in order to re-run the PUF with a new duration, in case of overflowing or insufficient counts. The root of trust reads $magnitude$ values via a mux, by setting `puf_select` and observing `puf_readout`. Afterwards, the root of trust disables the readout by setting `puf_readout`, which latches and forces `puf_readout` to `0xFF.F` until the processor is reset. At reset, counters and disabling latch are cleared, and the PUF is re-run, allowing a new readout.

We implement the LPN PUF hardware on a Xilinx Zynq 7000 FPGA by manually implementing a portable ring oscillator and counter circuit, and aggressively constraining the relative placement of its components. This implementation, as shown in Figure 9, occupies a *column* of `SLICEL` sites on the FPGA, which prevents the circuit to be placed sparsely, and ensures the ring oscillator and counter circuit are contiguous (constraining the circuit to a row would allow the FPGA tools to place the circuit sparsely across columns containing BRAMs, DSPs and other large blocks).

The ring oscillator is a chain of buffers implemented as `LUT6` elements. Through trial and error, we determined that using the `A6` input for the ring oscillator (inputs `A1`-`A5` drive a 5-input lookup table, and are not guaranteed glitch-free) yields the best results. The inverting element in the oscillator is implemented as an XOR of the enable signal and the feedback (`A1 A6 LUT6`

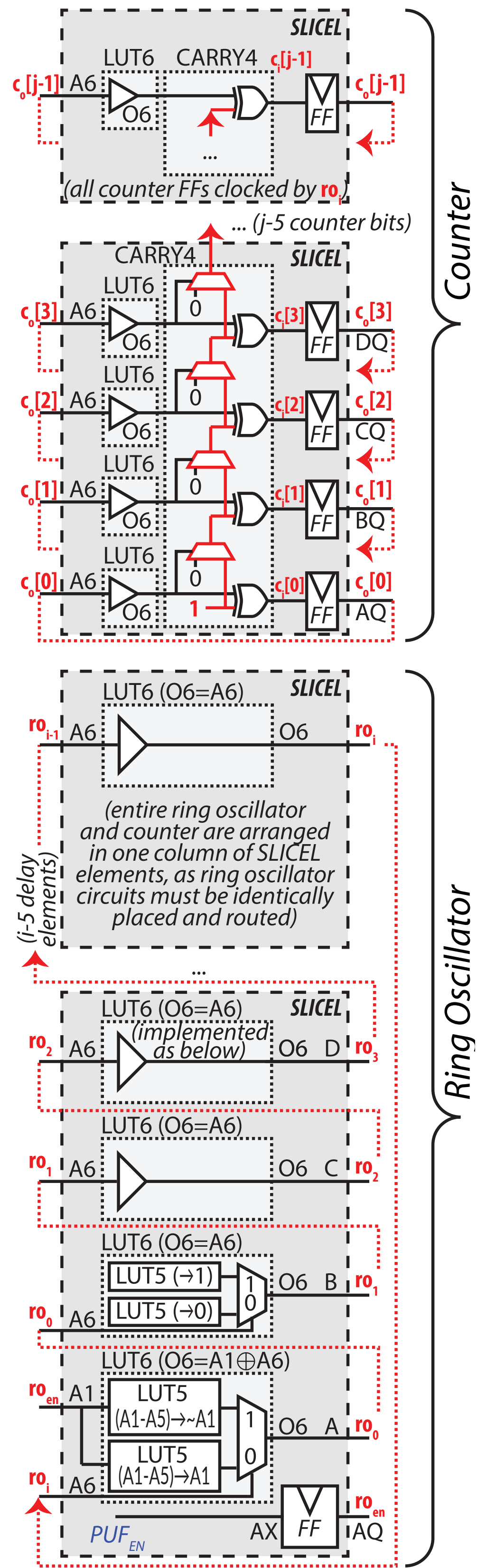

Fig. 9. Ring oscillator and counter, a critical component of the LPN PUF, as implemented on a Xilinx Zynq 7000 FPGA. The circuit is placed in the same column of `SLICEL` elements to ensure it is identically routed across all instances.

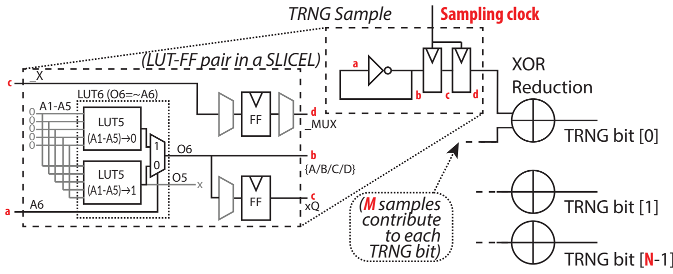

Fig. 10. A TRNG, as implemented on a Xilinx Zynq 7000 FPGA.

inputs, respectively). We constrain i delay elements of the ring oscillator to be laid out contiguously in a single column.

Adjacent to each ring oscillator is its counter circuit. As shown in Figure 9, we exploit detail of the `CARRY4` primitive on the FPGA to implement a small ripple-carry counter without the use of slow programmable logic. Like the ring oscillator, we constrain the j bits of counter to be laid out contiguously in the same column.

C. TRNG via jittery sampling of ring oscillators

In order to implement remote attestation for enclaves in the presence of malicious system software, Sanctum requires private, trustworthy entropy. This is needed, at a minimum, to complete a Diffie Hellman handshake to establish a secure communications channel with the trusted first party (cf. Section V). Enclaves that require non-deterministic behavior (such as enclaves that generate cryptographic keys) require additional entropy, but may rely on a PRNG seeded from the initial word of entropy.

Here, we describe a TRNG, which produces a stream of random data words by sampling free-running ring oscillators. Clock jitter is considered an unpredictable physical resource and, given that the (100MHz, as a generous maximum) sampling frequency is far less than the natural frequency of the short ring oscillator (in the gigahertz), we assume the relative jitter of these processes is sufficient to treat *subsequent* samples as independent and identically distributed (i.i.d.) random variables. Furthermore, we assume *different* (but identically implemented) ring oscillators and sampling logic to be i.i.d. Section VII-C rigorously tests these assumptions via a suite of statistical tests over the TRNG output.

While the bits sampled from the free-running ring oscillator do not have a 50% fair distribution (bias), an unfair coin flip can be "conditioned" toward fairness using a von Neumann corrector by instead recording the *parity* of several unfair (but i.i.d.) coin flips. To this end, the TRNG aggregates several (M) concurrently sampled ring oscillators which are XOR-ed to produce a single bit of output. Software is able to construct arbitrarily wide words of entropy by repeatedly sampling bits from the TRNG. To improve TRNG bandwidth and convenience, however, we concatenate a parallel vector of N such that sets of M ring oscillators produce N-bit words of entropy.

This structure, and the detail of placement and routing of one sampled ring oscillator in FPGA fabric is shown in Figure 10 and Section VII-C demonstrates that a cryptographically secure TRNG is achieved with $M = 7$ or better. To minimize bias, the ring oscillators and sampling circuit are identically placed and routed, by manually implementing the structure via constrained FPGA resources, as shown in the figure. In order to resolve metastability, each ring oscillator is doubly registered in the ring oscillator module. The output is aggregated with $(M-1)$ other ring oscillator samples (this circuit has no fanout), and latched in the core pipeline; three registers are widely considered sufficient to prevent metastable bits.

In Sanctum, the TRNG values are exposed to user-mode software (including enclaves) via a user-mode readable CSR (control/status register), allowing enclaves to access TRNG output without notifying the untrusted operating system. Each core implements its own TRNG to grant *private* entropy to each hardware process.

VII. Evaluation

We evaluate our approach to secure bootstrapping by measuring the code size and latency of the root of trust. The evaluation examines a variety of design points ($\mathbf{T}$, $\mathbf{P}$, $\mathbf{P_{AES}}$, $\mathbf{H}$, etc. defined in Section IV) addressing the threat model of a honest-but-curious manufacturer. The code size of the root of trust is a straightforward measurement of the required real estate in a processor's trusted boot ROM, which includes the devicetree and a rudimentary reset vector, as is the case with an insecure baseline.

Given the bootstrapping operation is performed in a trusted, isolated environment with no asynchronous events, we use the number of instructions executed by the root of trust program to estimate its latency (the time between a reset, and when the processor transfers control to the payload), given an assumed clock frequency. The in-order pipeline makes number of instructions a good proxy for latency. We augment this via a fixed cost model, which is used to approximate the overheads due to the memory hierarchy.

We separately examine our implementation of the required hardware subsystems: the TRNG and PUF, focusing on the process by which we select appropriate parameters for the security guarantees these primitives must provide.

We do not evaluate the remote attestation scheme as described in this manuscript, as it not a part of the root of trust (boot ROM), and is implemented in any of a variety of ways by the software system after boot. Also excluded from evaluation is the performance of inter-enclave communication and other aspects of the Sanctum processor, as these are not the main focus of this manuscript.

A. Root of Trust code size and latency

For all roots of trust considered, we evaluate a simulated system with 4 in-order 1GHz RISC-V cores booting a 128 KB payload. We model memory accesses via a simulated cache hierarchy consisting of a shared 4-way 1 MB L2 cache with 32 byte lines and a random replacement policy (3 cycle accesses),

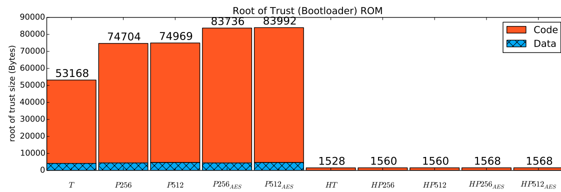

Fig. 11. Code size of variations of the root of trust.

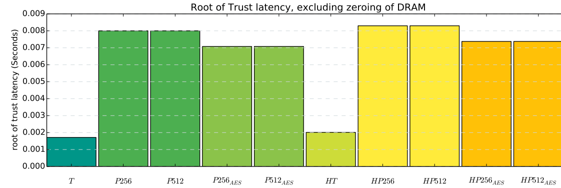

Fig. 12. Measurement root latency for variations on the root of trust, excluding the clearing of uninitialized DRAM.

and private data and instruction caches, each a set-associative 4-way 32 KB cache with 32 byte lines with a random replacement policy (30 cycle access time). Our modeled system has 1 GB DRAM and a 70 cycle access time.

We evaluate the six variations on the root of trust described in Section IV, and report the root of trust size (required trusted ROM size) in Figure 11. The minimal roots of trust ($\mathbf{HT}$ and similar) are small indeed, requiring only a 1.47 KB ROM, which includes only the code needed to copy a root of trust binary from untrusted memory, hash it, and compare the result against an expected constant. No significant variation in the size of the root of trust appeared with a varied size of the root of trust binary in untrusted memory, meaning $\mathbf{HT}, \mathbf{HP256}, \mathbf{HP256_{AES}}, \mathbf{HP512}, \mathbf{HP512_{AES}}$ all require a 1.5 KB ROM.

The corresponding latency of each root of trust considered is shown by Figure 11. As expected, the minimal root of trust ROM designs ($\mathbf{HT}$, etc.) incur a small increase (approximately 0.3 milliseconds) in root of trust latency, but exhibit an enormous reduction in boot ROM size.

This evaluation does not measure the time to *provision* the PUF (to generate a secret key and compute the corresponding public helper data), as this is a one-time operation, and is less complex than normal key recovery, which is included in the measurements here. Increasing the number of PUF ring oscillators (M) from 256 to 512 did not significantly increase the root of trust latency, as the costly matrix operations are performed on a 128 $\times$ 128 matrix in $GF(2)$ in all scenarios, and only the size of straightforward vector operations and linear scans are increased.

Storing AES-encrypted keys in untrusted memory is somewhat more efficient than re-generating the keys from their seeds at each boot, although this difference is dwarfed by the latency of erasing DRAM at boot, as discussed below.

The latency of sanitizing DRAM is *excluded* from these results, and is estimated to be 2.35 seconds (on our modeled system), via a straightforward software loop, for the 1GB of DRAM in the systems considered, or slightly less, if booting a larger payload. Given that the latency of sanitizing DRAM dominates the root of trust, all roots of trust considered exhibit approximately the same latency at under 2.4 seconds, with ample opportunity to accelerate the DRAM erasure via DMA operations. In a system with transparent encryption of DRAM, cryptographic erasure is a reasonable option (erase only the keys with which DRAM is encrypted, thereby making the encrypted data unrecoverable).

B. TRNG

In order to evaluate the TRNG implemented on an FPGA fabric, we sample a *contiguous* string of bytes from the TRNG (one sample per clock cycle), store the stream in a large memory, and inspect the resulting binary with a suite of statistical tests. The `dieharder` [33] suite of statistical tests is run on a gigabyte of contiguous bytes read from the TRNG; the test suite is invoked with default configuration (a true random string is expected to pass each test with 95% probability). The `ent` [42] test estimates bit bias (expected bit value) and entropy per bit in the same binary blob. We report the size of each TRNG configuration (measured by utilization of `SLICEL` resources), and a digest of the results of each test suite. The TRNG was constrained to densely pack the required circuit on the Xilinx Zynq 7000 FPGA fabric. A 4 inverter loop with two registers is packed into one `SLICEL` primitive, with additional `SLICEL`s used for an XOR reduction for a TRNG with $M > 1$.

TABLE I
TRNG PERFORMANCE AND COST.

M	**size** (`SLICEL`**s**)	**expected bit value**	**entropy per bit**	`dieharder` **tests passed**
1	16	0.5134	0.9995	15/114
3	64	0.5004	1.0000	55/114
5	96	0.5000	1.0000	75/114
7	136	0.5000	1.0000	**108**/114
9	176	0.5000	1.0000	**114**/114

The TRNG is configured to output a 64-bit word per cycle, and this evaluation considers a range ($\mathbf{M}$) of inverters XOR-ed together to produce each TRNG bit. Slight bias of individual bits translates into a reduced entropy – an effect mitigated by increasing $\mathbf{M}$ – the number of unpredictable bits XOR-reduced to produce one bit of entropy. Our evaluation (see Table VII-B) shows that a 64-bit TRNG produces a cryptographically secure random stream for M as low as 7 at a modest cost of FPGA resources (0.24% of the FPGA; for reference, a bare-bones in-order 32-bit processor without the cache subsystem weighs in at approximately 600 slices).

The TRNG can be significantly reduced in size by adjusting its N parameter to produce only a byte of entropy per sample. Furthermore, lesser parameterizations of M produce a reasonably high-quality random stream.

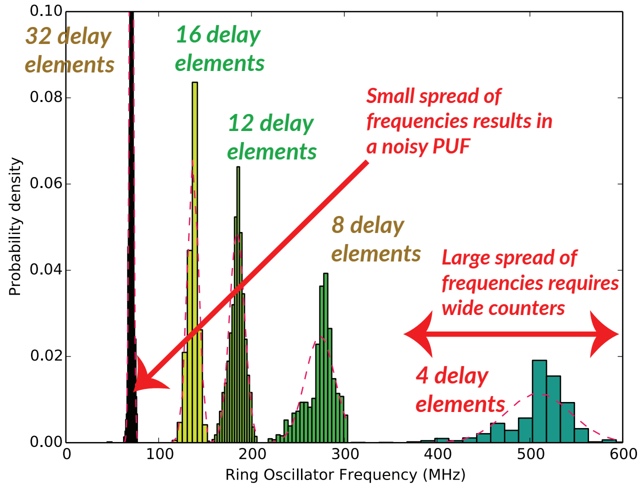

Fig. 13. Distribution of repeated frequency measurements for a population of **i**-element ring oscillators on a Xilinx Zynq7000 for several **i**: 4,8,12,16,32.

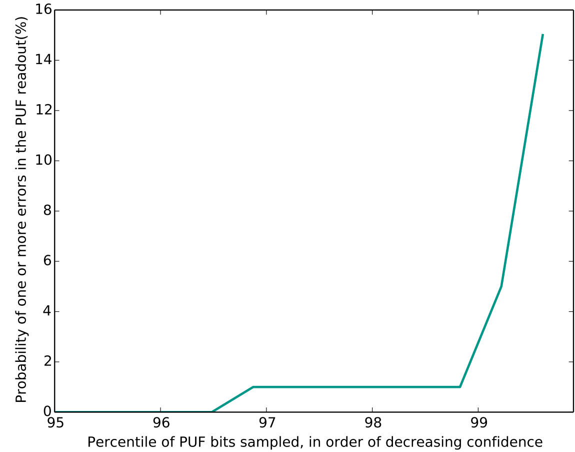

Fig. 14. Probability of one or more bit errors among a greedy sampling 512 ring oscillator pairs, in order of decreasing confidence.

C. Trapdoor LPN PUF

We examine the performance of the LPN PUF primitive in order to select an appropriate parameterization (M: number of ring oscillator pairs for $N = 128$, i.e., a 128-bit secret value) such that the processor is able to tolerate bit errors in typical conditions, and achieve negligible probability of failed key recovery at boot. All PUF measurements were performed with a Xilinx Zynq 7000 device (via a ZC706 development platform) in a typical setting, reasonably isolated from sources of electric interference, at 72 degrees F. We do not evaluate the repeatability of the ring oscillator pairs across temperature and power variations; prior work [20] demonstrates automotive variation in environmental conditions increases the noise exhibited by the PUF, and recommends a set of `450` ring oscillator pairs to obfuscate a 128-bit secret value.

Figure 13 shows a distribution of measured RO frequencies obtained from 1024 measurements spanning several hours and several power cycles of the FPGA platform. To measure this distribution, we implemented 1024 ring oscillators of various lengths driving a 12-bit counter. We run the ring oscillators for 5.12 microseconds (1024 200MHz processor cycles), and estimate ring oscillator frequency from recorded counter values (extremely fast ring oscillators may introduce errors in counts due to timing closure and overflow). We observe no significant counter glitches in the recorded population. The distribution of RO frequencies narrows significantly for oscillators with a longer delay line. Short, fast oscillators exhibit a wide distribution of frequencies, with some noise indicative of glitching.

Table VII-B details the distribution of ring oscillator frequencies for the 5 configurations considered, including the largest standard deviation in frequency of any single ring oscillator observed across the 1024 repeated samples. From this table and Figure 13, and based on trial and error, we note that an 8-gate ring oscillator is well-suited for an LPN PUF circuit: with a wide distribution of frequencies, it does not exceed the device's rated maximum, shows little evidence of glitching, and has little variation in RO frequencies over time.

TABLE II
RING OSCILLATOR PERFORMANCE AND COST.

delay elements	minimum frequency	expected frequency	maximum frequency	maximum deviation of RO frequency
4	307 MHz	515 MHz	608 MHz	9MHz
8	218 MHz	277 MHz	303 MHz	5MHz
12	96 MHz	185 MHz	205 MHz	2MHz
16	44 MHz	137 MHz	152 MHz	15MHz
32	62 MHz	70 MHz	76 MHz	0.6 MHz

Each PUF bit is a pair of ring oscillators and counters; the measurement of a PUF bit is the difference of the two counts, which conveys the bit *value* (the sign of the measurement), and *confidence* (the magnitude), as described in Section VI-B. To characterize the PUF bit error rate, we implement 512 PUF bits with 12-bit counters and 8-element ring oscillators. We enable the 512 PUF bits for a duration of 1024 processor cycles (5.12 microseconds), and read out the resulting magnitudes. We repeat this measurement 1024 times across several hours, occasionally power cycling the platform. The expected value of a PUF bit sampled from this population is `0.4688`. For each PUF bit, we define a "gold" value to be the median (most frequent) value across the 1024-sample population. Figure 14 shows the probability of one or more bit errors (relative to gold values) in one *entire* PUF readout. Under typical conditions, only about one percent of ring oscillator pairs we examined are unreliable. Among the 90% of ring oscillator pairs *with highest confidence value*, no errors were detected across 1024 readouts. Informally, we note that while this error rate is as high as

10-15% for other ring oscillator configurations, the confidence information remains a reliable means to select stable PUF bits.

We confirmed that 4 identical FPGA platforms produced different expected PUF outputs. For a population of 512 PUF bits, we observed 51 with the same value across all 4 platforms, in expectation, after 16 measurements. Under an i.i.d. assumption, we expect to observe 64 identical bits across 4 devices.

Prior work [20], [25] characterizes their PUF implementation across the automotive range of environmental conditions, and selects a much larger M (450) for the same $N = 128$ in order to tolerate a higher error rate. Given the above evaluation, we parameterize the LPN PUF with $M = 256$ for the $\mathbf{P256}$ root of trust (a generous margin of POK bits for a 128-bit key, given our observations under normal conditions), and $M = 512$ for the $\mathbf{P512}$ (informed by the reported increased noise in an automotive environment, as shown in prior work). In practice, the larger M considered in this case has little performance overhead, and requires only a few bytes of additional space in the on-chip boot ROM, as shown in Section VII-A.

VIII. Conclusion

We have provided a detailed description of the secure boot and remote attestation process in a prototype Sanctum processor. While a significant part of this paper was devoted to PUF-based key generation, the boot and attestation protocols are agnostic to where entropy comes from, and are equally applicable to the case where secret keys or seeds are stored in secure non-volatile memory, or simply generated by a TRNG. In our prototype implementation, we have shown that PUFs can be used to derive keys unknown to the manufacturer while providing an efficient boot and attestation process.

Acknowledgements

This work was partially funded by Delta Electronics, Analog Devices, and DARPA & SPAWAR under contract N66001-15-C-4066, and the DARPA SSITH program under contract HR001118C0018. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] "Trusted platform module library specification family "2.0"." Trusted Computing Group, 2014.
[2] Altera, "Secure Device Manager for Intel® Stratix 10 Devices Provides FPGA and SoC Security," `https://www.altera.com`.
[3] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," *Information Quarterly*, vol. 3, no. 4, pp. 18–24, 2004.
[4] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, vol. 13, 2013.
[5] *ARM Security Technology Building a Secure System using TrustZone® Technology*, http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, ARM Limited, Apr 2009, reference no. PRD29-GENC-009492C.
[6] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: SGX cache attacks are practical," *CoRR*, vol. abs/1702.07521, 2017. [Online]. Available: http://arxiv.org/abs/1702.07521
[7] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 132–145.
[8] J. Camenisch, M. Drijvers, and A. Lehmann, "Anonymous attestation with subverted tpms," in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 427–461.
[9] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, Feb 2016.
[10] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 857–874.
[11] ——, "Secure processors part II: Intel SGX security analysis and MIT sanctum architecture," in *FnTEDA*, 2017.
[12] S. Devadas and T. Ziola, "Volatile device keys and applications thereof," Jul. 21 2009, uS Patent 7,564,345.
[13] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: how to generate strong keys from biometrics and other noisy data," in *Advances in Cryptology - Eurocrypt 2004*, 2004.
[14] C. Fletcher, M. van Dijk, and S. Devadas, "Secure Processor Architecture for Encrypted Computation on Untrusted Programs," in *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing; an extended version is located at http://csg.csail.mit.edu/pubs/memos/Memo508/memo508.pdf (Master's thesis)*, Oct. 2012, pp. 3–8.
[15] B. Fuller, X. Meng, and L. Reyzin, "Computational fuzzy extractors," in *Advances in Cryptology-ASIACRYPT 2013*. Springer, 2013, pp. 174–193.
[16] B. Gassend, "Physical random functions," Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., Jan. 2003.
[17] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM conference on Computer and communications security (CCS)*, 2002.
[18] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996. [Online]. Available: citeseer.nj.nec.com/goldreich96software.html
[19] D. Grawrock, *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.
[20] C. Herder, L. Ren, M. van Dijk, M.-D. Yu, and S. Devadas, "Trapdoor computational fuzzy extractors and stateless cryptographically-secure physical unclonable functions," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, pp. 65–82, 2017.
[21] M. Hiller, D. Merli, F. Stumpf, and G. Sigl, "Complementary IBS: Application Specific Error Correction for PUFs," in *IEEE Int. Symposium on Hardware-Oriented Security and Trust*. IEEE, 2012.
[22] M. Hiller, M. Weiner, L. Rodrigues Lima, M. Birkner, and G. Sigl, "Breaking Through Fixed PUF Block Limitations with Differential Sequence Coding and Convolutional Codes," in *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, ser. TrustED '13, 2013, pp. 43–54.
[23] D. Holcomb, W. Burleson, and K. Fu, "Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags," in *Proceedings of the Conference on RFID Security*, Jul. 2007.
[24] T. Hudson, "Heads," 2017. [Online]. Available: https://trmm.net/Heads
[25] C. Jin, C. Herder, L. Ren, P. H. Nguyen, B. Fuller, S. Devadas, and M. van Dijk, "Fpga implementation of a cryptographically-secure puf based on learning parity with noise," *Cryptography*, vol. 1, no. 3, 2017. [Online]. Available: http://www.mdpi.com/2410-387X/1/3/23
[26] kokke, "tiny-aes-c," https://github.com/kokke/tiny-AES-c, 2018.
[27] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanovic, "A 45nm 1.3 GHz 16.7 double-precision GFLOPS/w RISC-V processor with vector accelerators," in *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*. IEEE, 2014, pp. 199–202.

[28] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[29] R. Maes, P. Tuyls, and I. Verbauwhede, "Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2009, pp. 332–347.

[30] ——, "Soft Decision Helper Data Algorithm for SRAM PUFs," in *Proceedings of the 2009 IEEE International Conference on Symposium on Information Theory - Volume 3*, ser. ISIT'09, 2009, pp. 2101–2105.

[31] R. Maes, A. Van Herrewege, and I. Verbauwhede, "PUFKY: A Fully Functional PUF-based Cryptographic Key Generator," in *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES'12, 2012, pp. 302–319.

[32] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," *HASP*, vol. 13, p. 10, 2013.

[33] G. Novark and E. D. Berger, "Dieharder: Securing the heap," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 573–584. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866371

[34] O. Peters, "Ed25519," https://github.com/orlp/ed25519, 2018.

[35] X. Ruan, *Boot with Integrity, or Don't Boot*. Berkeley, CA: Apress, 2014, pp. 143–163. [Online]. Available: https://doi.org/10.1007/978-1-4302-6572-6_6

[36] M.-J. O. Saarinen, "tiny_sha3," https://github.com/mjosaarinen/tiny_sha3, 2018.

[37] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2435–2450.

[38] G. E. Suh, "Aegis: A single-chip secure processor," Ph.D. dissertation, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., Aug. 2005.

[39] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th annual international conference on Supercomputing*. ACM, 2003, pp. 160–171.

[40] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *ACM/IEEE Design Automation Conference (DAC)*, 2007.

[41] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions," in *Proceedings of the* 32nd *ISCA'05*. New-York: ACM, June 2005. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-483/Memo-483.pdf

[42] J. Walker, "A pseudorandom number sequence test program," http://www.fourmilab.ch/random/, 2018.

[43] W. C. Waterhouse, "How often do determinants over finite fields vanish?" *Discrete Mathematics*, vol. 65, no. 1, pp. 103 – 104, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0012365X87902172

[44] A. Waterman, K. Lee, Asanovic, and S. Inc., "The RISC-V instruction set manual volume II: Privileged architecture version 1.10," EECS Department, University of California, Berkeley, Tech. Rep., May 2017. [Online]. Available: https://riscv.org/specifications/privileged-isa/

[45] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[46] R. Wojtczuk and J. Rutkowska, "Attacking Intel trusted execution technology," *Black Hat DC*, 2009.

[47] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Another way to circumvent Intel® trusted execution technology," *Invisible Things Lab*, 2009.

[48] Xilinx, "Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices," `https://www.xilinx.com`.

[49] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 640–656.

[50] M.-D. M. Yu and S. Devadas, "Secure and robust error correction for physical unclonable functions," *IEEE Design and Test of Computers*, vol. 27, pp. 48–65, 2010.