This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Hardware Assisted Buffer Protection Mechanisms for Embedded RISC-V

Asmit De, Aditya Basu, Swaroop Ghosh and Trent Jaeger

***Abstract*—RISC-V is a promising open source architecture that targets low-power embedded devices and SoCs. However, there is a dearth of practical and low-overhead security solutions in the RISC-V architecture. Programs compiled using RISC-V toolchains are still vulnerable to code injection and code reuse attacks such as buffer overflow and return-oriented programming (ROP). In this paper, we propose two hardware implemented security extensions to RISC-V that provides a defense mechanism against such attacks. We first employ a Physically Unclonable Function (PUF)-based randomized canary generation technique that removes the need to store the sensitive canary words in memory or CPU registers, thereby being more secure, while incurring low overheads. We implement the proposed Canary Engine in RISC-V RocketChip with Rocket Custom Coprocessor (RoCC). Simulation results show 2.2% average execution overhead with a single buffer protection, while a 10X increase in buffer count only increases the overhead by 1.5X when protection is extended to all buffers. We further improve upon this with a dedicated security coprocessor FIXER, implemented on the RoCC. FIXER enforces fine-grained control-flow integrity (CFI) of running programs on backward edges (returns) and forward edges (calls) without requiring any architectural modifications to the processor core. Compared to software-based solutions, FIXER reduces energy overhead by 60% at minimal execution time (1.5%) and area (2.9%) overheads.**

***Index Terms*— Buffer overflow, PUF, Stack Canary, RISC-V**

I. Introduction

Programming languages such as C which are closer to the hardware provide a lot of flexibility in terms of memory and IO access to allow system and device level programming. However, such languages are weakly typed and often tend to have inherent deficiencies leading to security vulnerabilities if not used with proper and secure practices. Buffer overflow (Fig. 1) is the most common vulnerability that can be exploited to launch a variety of attacks. In a program without bounds checking, an adversary can overload a user input with excess data that can overrun the buffer capacity and overwrite nearby memory locations with potentially malicious data, leading to attacks such as return-oriented programming (ROP), function pointer manipulation and violation of data flow integrity.

Stack canaries [4] are *sacrificial* words placed on the stack at

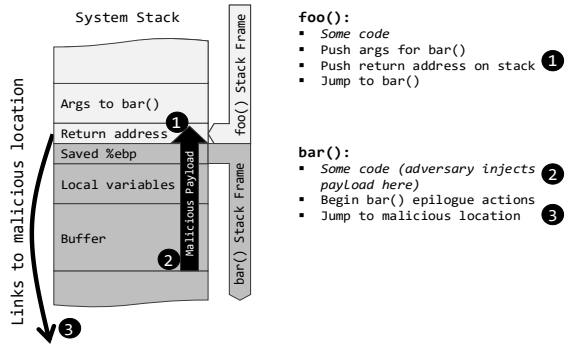


Fig. 1. Buffer overflow exploit.

stack frame boundaries to detect potential return address overwriting. If an adversary overflows a buffer in order to overwrite the return address, the canary is also overwritten. Before returning in the program's execution stack, the canary is checked, and if modified, the return address is assumed to be compromised. This approach works if the adversary's target is to overflow a buffer to overwrite the return address. However, there are scenarios where the adversary can skip over the canary using a vulnerable pointer reference, guess the canary, or learn the canary using a disclosure vulnerability.

Unfortunately, stack canaries cannot detect a buffer overflow if the attack payload does not actually overwrite the canary value. In a data-oriented attack, the adversary can overflow the buffer just enough to overwrite some sensitive variable above the buffer, but not cross stack frame boundaries. Here, the canary will not be overwritten; hence, it will not be able to detect the attack. Fig. 2 shows a typical layout of the stack frame, where such an attack is possible.

Fig. 3 shows an example of vulnerable code for a data-flow attack [5]. In this example, an adversary can send more than 1000 bytes of data which the `PacketRead` function writes to the `packet` variable. If the adversary's payload is large enough, it will overwrite the return address which will be detected by the canary when the stack rolls back. However, if the payload is carefully crafted, adversary can just overwrite the `authenticated` variable above the `packet` buffer. Then the check `Authenticate(packet)` can be bypassed and the packet will be processed, without being detected by the canary.

Asmit De is with The Pennsylvania State University, University Park, PA 16802 USA (email: asmit@psu.edu).

Aditya Basu is with The Pennsylvania State University, University Park, PA 16802 USA (e-mail: aditya.basu@psu.edu).

Swaroop Ghosh is with The Pennsylvania State University, University Park, PA 16802 USA (e-mail: szg212@psu.edu).

Trent Jaeger is with The Pennsylvania State University, University Park, PA 16802 USA (e-mail: trj1@psu.edu).

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

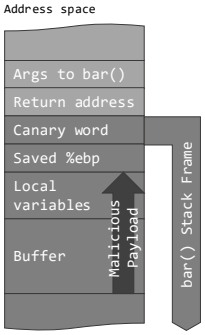


Fig. 2. A vulnerable stack frame layout with stack canaries.

```
1.  int authenticated = 0;
2.  char packet[1000];
3.
4.  while (!authenticated) {
5.      PacketRead(packet);
6.
7.      if (Authenticate(packet))
8.          authenticated = 1;
9.  }
10.
11. if (authenticated)
12.     ProcessPacket(packet);
```

Fig. 3. A vulnerable C code.

The obvious solution is to detect a buffer overflow as soon as it happens, and not wait for the function to end and the stack to rollback in order to validate the canary. One possible way is to place canaries at the top of every buffer. This leads to some challenges. Canaries are held in specialized canary registers or in a protected memory location in the address space. If the canaries are randomized, they take up some memory space or several registers. This is expensive in terms of memory space, especially if we try to put multiple canaries in the same stack frame to protect every buffer. Moreover, a vulnerability in saved canary locations can also lead to disclosing the canaries. The canaries also need to be validated after every write to a buffer. This can be expensive if implemented in a fine-grained manner. Existing solutions to protect programs from data-flow integrity are software based, e.g., performing reaching definitions analysis [5], or enforcing compile-time memory safety constraints [48], while others use specialized hardware or architecture to perform tagging and metadata processing [6-7]. However, these techniques are expensive in terms of performance and/or memory requirements or require hardware or architectural modifications to support them.

In this work, we present PUFCanary, a fine-grained yet lightweight hardware generated stack canaries to protect buffer boundaries and can detect overflow of the buffers using Physically Unclonable Function (PUF) [8]. The PUF generates randomized canary words in a secure manner based on the address in use. We implement our design in RocketChip [9] based on the RISC-V architecture. The Rocket Custom Coprocessor (RoCC) of RocketChip allows a flexible hardware design implementation of our Canary Engine without modifying the core processor architecture. Compared to existing stack canary, our design provides the following key benefits: (i) secure and randomized canary word generation using PUFs, (ii) fine-grained individual buffer protection, and, (iii) no need to save canaries in memory/ registers.

We further design FIXER (Flow Integrity Extensions for Embedded RISC-V), a low energy, low overhead security coprocessor that ensures integrity of backward and forward edge control flow of programs running on a RISC-V core. FIXER decouples the security architecture from the RISC-V core architecture, enabling a highly flexible security design. In the target deployment platform, the unmodified RISC-V core will be a hard IP, while the dynamically reconfigurable FIXER coprocessor will be implemented on an on-chip FPGA. Such an approach has the potential to be scaled to hybrid processor designs e.g., a Xeon + FPGA core [10]. The FPGA also provides the flexibility to change and update the security architecture in demand to new threats, without a complete redesign of the primary computing core. With the number of vulnerabilities rapidly increasing, it demands an efficient low-power flexible and scalable security solution that is sustainable for long periods of time. FIXER potentially unlocks the design capability to protect our systems from such cybersecurity threats. Software based CFI techniques are also limited by the size of the address space, which can be overcome by FIXER's flexible FPGA implementation. Compared to NILE [11], FIXER achieves better performance. Although NILE uses an unmodified RISC-V core similar to FIXER, the core-coprocessor interface is modified for the coprocessor to tap into more resources of the core. Note that, even though PUFCanary and FIXER both aim to protect memory, they approach the solution in different ways. PUFCanary tries to proactively detect one of the fundamental causes of memory exploits – the buffer overflow itself. This requires compiler support and incurs more overhead than FIXER, especially if protecting multiple buffers. However, this allows protection against both control-flow and data-flow attacks originating from buffer-overflow. Hence security critical systems may opt to use PUFCanary, trading in some performance. FIXER, on the other hand, implements a shadow stack and policy memory for preventing control flow violations, but it cannot prevent data-flow attacks. The tradeoff is simpler design and better performance than PUFCanary, hence this can be used for resource-constrained embedded systems. Table I shows a qualitative comparison of PUFCanary and FIXER with the state-of-the-art memory protection solutions. Both PUFCanary and FIXER maintain high-performance with low energy, the difference being PUFCanary can also detect data-flow attacks. PUFCanary and FIXER are both hardware agnostic, however, special compiler support is required for PUFCanary implementation. FIXER also has the added benefit of being dynamicaly updated due to it's flexible FPGA implementation.

The major contributions of this work are: (a) a secure PUF

TABLE I. Qualitative Comparison of FIXER with Related Works

	Canary [4]	ASLR [3]	CFI [1]	PUMP [18]	HAFIX [27]	GRIFFIN [29]	HDFI [31]	NILE [10]	**PUFCanary**	**FIXER**
Control flow hijacking protection	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Data flow hijacking protection	✗	✓	✗	✓	✗	✗	✓	✗	✓	✗
Maintains high-performance	✗	✗	✗	✗	✓	✗	✓	✓	✓	✓
Low energy overhead	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
No architecture modifications	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓
No source code pre-processing	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗
No compiler modifications	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓
Software flexibility	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Hardware flexibility	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Dynamic patching	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

based hardware generated canary design; (b) fine-grained individual buffer protection using canaries; (c) PUF design optimizations to improve performance.

The paper is organized as follows: Sections II and III describe existing defense mechanisms, RocketChip and the Rocket Custom Coprocessor architecture. Section IV presents the design flow and implementation of the PUF-based randomized canaries. Section V details the FIXER security coprocessor architecture. Limitations are discussed in Section VI, and conclusions are drawn in Section VII.

II. Defense Mechanisms

Stack Canaries: Stack canaries [4] are sacrificial words placed on the stack at stack frame boundaries to detect potential return address overwriting. If an adversary overflows a buffer and overwrites the return address, the canary gets overwritten. Before returning in the execution stack, the canary is checked, and if modified, the return address is assumed to be compromised, and the program is halted.

Data Execution Prevention: Data Execution Prevention (DEP) [2] prevents an adversary from executing malicious code from the stack. Memory pages are marked W⊕X, meaning, a page can either be executable (code) or be writable (stack, heap), but not both. However, an adversary can return to existing code in the program or shared libraries using gadget chains (return-to-libc attack).

Address Space Layout Randomization: Address Space Layout Randomization (ASLR) [3] randomizes the code, stack, heap, and shared library locations on the address space, to make it difficult to determine specific addresses and launch attacks. However, buffer overread and side-channel vulnerabilities can be used to reverse engineer the randomized address.

Control Flow Integrity: Control Flow Integrity (CFI) [1] involves statically computing a valid control flow graph (CFG) of the program and ensuring that during runtime, the program abides by that CFG. A coarse-grained approach to ensuring CFI while returning from functions is the use of a shadow stack (a separate stack residing in a secure memory location) [12]. On each function call, the return address is saved on the shadow stack alongside being put on the stack normally. While returning from a function, the return address on the stack is validated against the one on the shadow stack. On mismatch, it is assumed that the return address has been compromised. However, a shadow stack can be performance intensive since the pages housing the shadow stack may not be present in cache and may require several cycles to bring the page onto the cache and perform the validation. Several software techniques have been proposed for supporting shadow stacks [13-14].

Even with the presence of a shadow stack, an adversary can bend the control flow of a program. To prevent such incorrect control flows for indirect calls, the program is first analyzed to compute a coarse-grained or fine-grained CFG [1]. A policy matrix can then be created from the CFG that specifies the allowed call targets for each call site. During execution, for each indirect call, the policy matrix is looked up to determine the validity of the call target. However, this approach still suffers from similar performance degradation if the policy resides in memory. Compile-time and runtime enforcement of CFI have been shown in [15-16]. Lazy CFI [17-18] can somewhat alleviate the performance loss, but that leaves room for generating false negatives.

Secure hardware platforms: ARM TrustZone [19] and Intel SGX [20] isolate the hardware restrict access to systems assets. Hardware acceleration of security validation has been proposed to address the performance impact partially while covering a subset of security threats e.g., Intel CET [21] to protect against control-flow hijacking. Intel MPX [22] is developed to prevent memory safety violations. Intel TSX [23] exposes and exploits hidden concurrency in multi-threaded applications. Intel PT [24] logs TSX events when a transaction begins, commits or aborts. It has been shown in [25] that tagging of code and data using software-defined metadata and processing the tag using custom designed processor can detect ROP, code injection, memory safety violation and pointer corruption. Although effective, this new architecture cannot be readily deployed due to lack of re-configurability, and, area, energy and performance overheads. Other hardware-assisted techniques to enforce CFI are proposed in [25-29]. Data flow protection in stack and heap using hardware assistance is also proposed [30-31]. Specialized hardware stack redundancy systems have been developed for embedded systems [32-35], however these are architecture dependent and cannot be updated post-deployment. The common challenges associated with these secure hardware platforms include design overhead, lack of provisions to patch the design and keep pace with rapidly evolving threats, need of program binary instrumentations, compiler modifications, and, lack of adaptability to adjust the security level in runtime as needed. To alleviate these issues, a decoupled architecture using hardware performance monitors implemented on a RISC-V coprocessor has been proposed in [11].

III. Overview of the RocketChip Architecture

The PUFCanary and FIXER architectures are based on Rocket Chip [9] (written in CHISEL [36]), an open source parameterized system-on-chip (SoC) design generator. We use the RocketChip generator to generate synthesizable RTL for the standard Rocket Core SoC, a six-stage single-issue in-order pipeline processor that executes the 64-bit scalar RISC-V ISA (Fig. 4(a)). The Rocket Tile consists of the scalar core, the L1 caches, and the Rocket Custom Coprocessor (RoCC). The RoCC is a user-defined accelerator for the core which communicates with core over the RoCCIO interface using a set of custom instructions.

RoCC Instructions: The 32-bit RoCC instructions extend the RISC-V ISA and are encoded as shown in Fig. 4(b). The four custom instructions supported by Rocket Chip is shown in Table II. The *xs1*, *xs2*, and *xd* bits control read and write of the core registers by the RoCC instruction. If *xs1* is 1, then the 64-bit value in the integer register specified by rs1 is passed to the RoCC. If the *xs1* bit is clear, no value is passed over the RoCCIO interface. Similarly, *xs2* bit controls the read of register specified by *rs2*. If the *xd* bit is 1 and *rd* is not 0, the

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

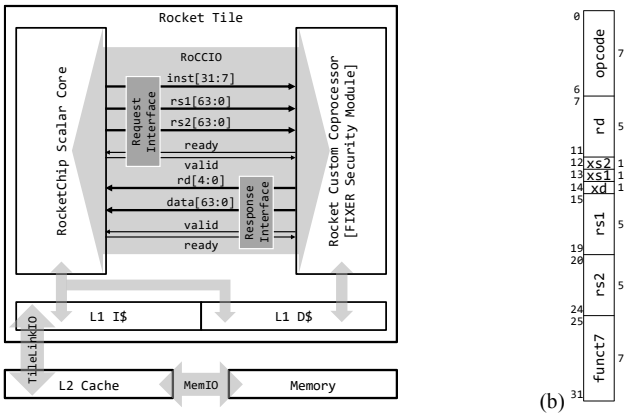


Fig. 4. (a) RocketChip architecture. FIXER coprocessor is also shown, (b) RoCC instruction encoding.

TABLE II. RoCC Instruction Opcodes

RoCC Instruction	Opcode
custom0	0001011
custom1	0101011
custom2	1011011
custom3	1111011

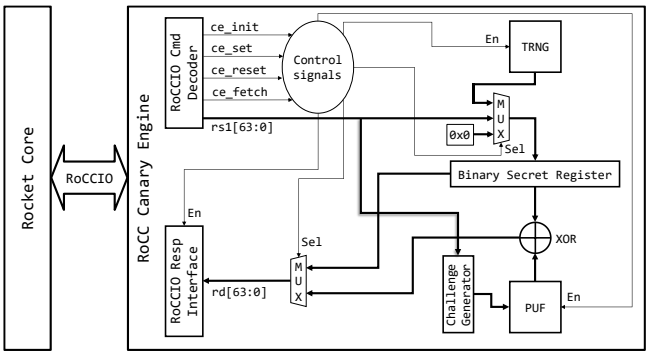


Fig. 5. Canary Engine design.

core will wait for a value to be returned by the coprocessor over the RoCCIO after issuing the instruction to the coprocessor. The value is then written to the register specified by *rd*. If the *xd* is 0 or *rd* is 0, the core will not wait for a value from RoCC. The *opcode* field specifies the custom instruction for the RoCC, and the *funct7* field further specifies a user-defined function implemented in the RoCC. The RoCC is responsible for signaling illegal instructions to the core.

RoCCIO Interface: The RoCC interacts with the Rocket core and the shared memory system via the RoCCIO interface (Fig. 4(a)). The core initiates a RoCC command by passing the RoCC instruction to the coprocessor via *inst*, as well as the relevant register values via *rs1* and *rs2*. If the RoCC instruction has the *xd* bit set, then the RoCC must eventually supply a response value over the RoCC response interface via *data*.

IV. PUF Based Randomized Canaries

A. Physically Unclonable Function (PUF)

PUFs are a secure hardware fingerprint applied in hardware crypto systems. PUF generates the response (key) to a particular challenge from physical properties of the chip. The exhaustive set of challenge-response pairs (CRP) serves as the fingerprint of the PUF chip, which is fixed (even across power cycles) for a particular chip but varies chip-to-chip. Several flavors of PUFs exist in literature [37-39], etc. Specially crafted circuit structures e.g., SRAM and flip-flops are used to amplify physical randomness for the PUF signature. Traditionally, PUFs have been used for chip authentication and deter IC counterfeiting. In this work, we use SRAM PUF for ease of implementation however, other types of PUFs can also be used.

B. True Random Number Generator (TRNG)

TRNG harnesses the natural entropy present in the system e.g., thermal noise [40], shot noise, Brownian motion or nuclear decay [41]. Techniques to harvest the noise in the operational amplifier [42], jitter of coupled oscillators [43], state of bi-stable elements [44] and oxide breakdown of transistors [45] have also been proposed. The challenges involved in designing TRNG include exploiting new entropy sources, efficient harvesting mechanisms and careful post-processing. In this work, we use FPGA's oscillator jitter for TRNG, although other variants can also be used.

C. Generating Randomized Canary Words

In our proposed design methodology, we generate and place one canary per buffer in the program's execution stack. This is in contrast to the standard canary implementation where only one canary is placed at the return boundary of an execution stack. Furthermore, we randomize the canaries such that all the canary words in use for the current process are unique. This is to mitigate any attacks resulting from disclosure vulnerabilities. We design a Canary Engine using a PUF and a TRNG (Fig. 5) to generate random canaries. Since the canaries are placed at specific locations in a program's address space, we randomize the canaries based on the address where the canary will be placed. The PUF in the Canary Engine works in challenge-response mode, where the *address location for the canary is used as challenge*. The PUF response r_a is a partial canary word based on the challenge address a. The $\{a \rightarrow r_a\}$ mapping is obtained from the PUF signature (CRP). This partial word is used in conjunction with a binary secret value s, kept in a dedicated binary secret register in the Canary Engine. This is crucial in order to make the canary word truly unpredictable for the adversary, since the PUF by itself is not a secret due to its fingerprint nature. We use the TRNG in the Canary Engine to create the secret value s_p for a process p when it is spawned for the first time. The secret value needs to remain constant for the lifetime of a process, hence it is backed up in the process's Process Control Block (PCB) by the operating system, so that it can re-populate the register with the value when the process is switched back in (context switch). The partial word r_a from the PUF is XORed with the binary secret s_p to generate the final canary word w for the requested address a:

$$w = r_a \oplus s_p \quad (1)$$

Note that, in the canary generation procedure, PUF serves the purpose of randomization within the same process, while the binary secret makes the canary unique across processes.

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

(a)

```
00007ff3ee05b000     24K rw---   [ anon ]
00007ff3ee078000      4K r---- ld-2.23.so
00007ff3ee079000      4K rw--- ld-2.23.so
00007ff3ee07a000      4K rw---   [ anon ]
00007ffc0bcd3000    132K rw---   [ stack ]
00007ffc0bd90000     12K r----   [ anon ]
00007ffc0bd93000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total             20840K
```

(b)

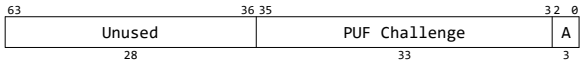


Fig. 6. (a) A partial `pmap` result of a process showing the addresses used by the stack, (b) the address bits chosen for the PUF challenge ('A' represents the 3 unused alignment bits).

PUF optimization: A 1:1 PUF mapping produces a unique response for every challenge. In a 64-bit architecture, we have 2^{64} addressable locations, where, segments such as kernel space and code segment will not be writable and will never be used for canary placement. Moreover, if a word is 8 bytes, and the data is word aligned, only 2^{61} addresses will actually be used for addressing and the lower order 3 bits can be ignored. To optimize our design for this, we can use part of the address bits as challenge instead of all 64 bits. For example, the higher order bits (kernel space) and the lower order bits (for alignment) can be ignored. Fig. 6(a) shows a partial `pmap` output for a 64-bit process. We can see that the stack starts at address `0x7ffc0bcd3000` and can grow up to `0x7ff3ee07a000` from where the memory mapping segment begins. Hence, we can practically ignore the higher order 28 bits (conservatively). Fig. 6(b) shows the chosen bits from an address for the PUF challenge. However, if this is not chosen conservatively, it can lead to duplicate challenges. It should be noted that the chosen bits are architecture specific and needs to be tuned for different architectures, which determines the size and region of memory that can be protected. Furthermore, this optimization is only applicable for user-space stack buffer protection only. For kernel space protection, the higher order addresses are also required, hence there is less scope of optimization. For protecting the entire address space, other physical optimization measures can be taken, such as downsizing the PUF.

D. Security Benefits and Implications

Our canary design system provides several benefits over the conventional canary implementation. We can generate multiple canaries in the same stack frame, protecting each buffer in the stack frame. Due to PUF usage, the canaries need not be saved in a register or in the address space (typically referenced by an offset from the x86 segment register *%gs* in GCC). The PUF signature (CRP) itself acts as a repository for the canaries to use. Whenever the canary at a particular address needs to be placed or validated, the PUF can be queried with the address as the challenge, and it will respond with the correct partial canary word. This makes it more secure and less prone to disclosure.

The PUF response is XORed with the binary secret to eliminate a brute-force disclosure of all the canaries from the PUF. Since the PUF signature is fixed over multiple power cycles, it is not a strong secret by itself. It generates different responses for each challenge, but for the same challenge, it generates the same response every time. This holds for challenges within the same process, across multiple executions of the same process, and even across different processes. Note that the responses change chip-to-chip. *Therefore, a successful attack on one system cannot be deployed globally*. Without the binary secret in place, an adversary can generate all possible challenges (addresses) and retrieve corresponding responses (canaries) for those addresses. Thus, the binary secret allows us to obfuscate the PUF response. To generate a truly random value for the secret, the hardware TRNG (much faster than software pseudo-random generator) is used. This provides a far more efficient and secure random number as the secret value for each invocation of the same program, or for different programs. This ensures that an adversary cannot re-compute the canary values using the secret.

We assume that the OS kernel is secure, and the PCB information cannot be disclosed from the user space. This is important, since the binary secret is a critical information for the particular process and must be stored securely in the PCB of the process in the kernel space to handle context switches. We also assume that the Canary Engine is secure, i.e., there is no instruction that can be used to directly query the PUF and obtain the response. Hence, there is no direct way of obtaining the exhaustive CRPs of the PUF. Also, there are no unprivileged instructions to read the binary secret register, preventing any information leakage from the Canary Engine that can be potentially leveraged to obtain the secret value in order to reconstruct the canaries. Since the secret value is 64-bit wide, brute-forcing it will require 2^{64} tries. For further securing the secret, it can be encrypted before storing in the PCB, however this may increase context-switching time due to the encryption/decryption process.

For a particular process p_1, the different canary words will be generated as $w_1 = r_{a_1} \oplus s_{p_1}, w_2 = r_{a_2} \oplus s_{p_1}$, etc. following (1). In case of a disclosure vulnerability, if w_1 is known for address a_1, it is not possible for the adversary to compute w_2 for a_2, since neither r_{a_1} nor s_{p_1} can be individually determined.

In our implementation, we have performed the XOR operation of the binary secret value with the PUF response. However, the secret can also be XORed with the address and used as PUF challenge. Either cases provide the same security guarantees. In both cases, the raw CRPs for the PUF remain undisclosed to the adversary due to the XOR operation, since only the challenge or the response is transparent to the adversary, but never both. To completely hide the PUF signature, the XOR operation can be performed on both sides, however, it will reduce performance. The number of PUF CRPs needs to be more than the size of the address space used for canary placement to avoid collisions in canary words.

E. Canary Usage and Design Flow

A system with our proposed canary design will be modified as follows: The kernel scheduler is modified to include a few extra instructions to configure the Canary Engine for the

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

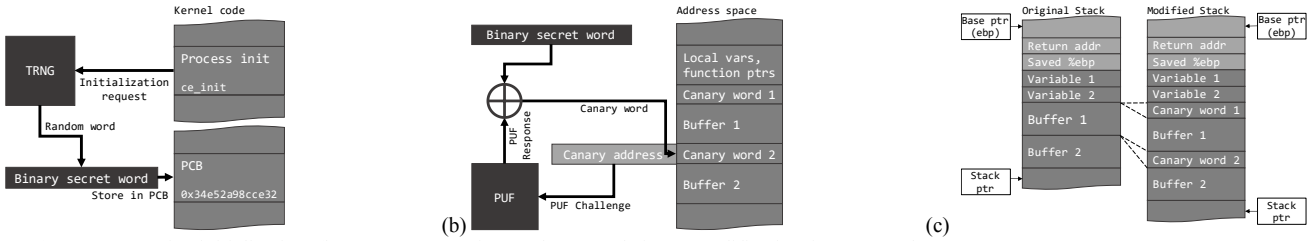


Fig. 7. (a) Canary Engine initialization, (b) Canary generation, and, (c) Stack frame modification for canary placements.

protection enabled process. Initially, when the process is to be scheduled for the first time, the kernel sends `ce_init` instruction to the Canary Engine. This is a privileged instruction and cannot be used from the user space. The TRNG in the Canary Engine generates the random word and populates the binary secret register. It also sends the value back to the kernel. The kernel saves the value in one of fields of the PCB for the particular process. This is shown in Fig. 7(a).

During a context switch, when the protected process is to be switched out, the kernel sends `ce_reset` instruction (privileged) to the Canary Engine. After decoding, a control signal is sent to the multiplexer to write a `0x0` (reset) value to the binary secret register to prevent disclosure of the secret.

When the process is about to be scheduled again, the kernel sends `ce_set` instruction (privileged) to the Canary Engine along with the secret word stored in the PCB. This writes the secret word and repopulates the binary secret register.

The program to be protected with canaries is compiled as follows: A static analysis is performed on the assembly code to identify the buffer locations in each function. A `ce_fetch` instruction is placed per buffer in the function prologue. This is the only unprivileged instruction that can be sent to the Canary Engine. The instruction sends the address of the canary location to the Canary Engine which decodes the address and sends the challenge to the PUF. The PUF responds with the partial canary word which is XORed with the binary secret register value to generate the final canary word. The canary word is sent back to the process to be placed on the memory location. The canary generation process is depicted in Fig. 7(b).

Placing the canaries on the stack frame also requires some readjustment of the stack boundaries due to the extra memory locations taken up by the canaries. This is accomplished by readjusting the stack pointer and base pointers. Furthermore, since the original stack layout has been altered, the locations of the variables and buffers in the stack also needs to be readjusted. This is done by changing the references to all the variables and buffers in the stack after taking into consideration the canary placements. The unmodified and altered stack frames are shown in Fig. 7(c).

F. Canary Engine Implementation using RoCC

The Canary Engine implementation in the RoCC is shown in Fig. 5. The program binary runs on the Rocket Core and sends RoCC instructions over the RoCCIO whenever a canary generation or validation is required. The RoCC instruction is first passed through the Cmd decoder, which extracts the individual fields of the RoCC instruction, and the contents of the two registers *rs1* and *rs2* if specified. The opcode field is decoded to the *custom0* instruction in our implementation. The *funct7* field is decoded to interpret `ce_init`, `ce_set`, `ce_reset` and `ce_fetch` instructions.

When the kernel sends the `ce_init` instruction to the Canary Engine, after decoding, a control signal is sent to the TRNG to generate a 64-bit random word. The word is sent to a multiplexer input. The appropriate select signal is sent to a multiplexer to select and write the random word to the binary secret register. At the same time, the word is sent to another multiplexer to be written to the *rd[63:0]* response interface and sent back to the core register *t0* to be saved in the process PCB.

When decoding the `ce_reset` instruction, a control signal is sent to the multiplexer, which selects the 0x0 (reset) value and writes it to the binary secret register. This resets the Canary Engine when the protected process is not in context.

For a `ce_set` instruction, the contents of the *t0* (the secret value held in the PCB of the process) is sent through the *rs1[63:0]* field of the RoCCIO interface to the canary engine. After the instruction is decoded, the value is read from the *rs1[63:0]* field and sent to a multiplexer. The required select signals are sent to the multiplexer and the value is copied to the binary secret register. This reconfigures the Canary Engine for the current process.

For the `ce_fetch` instruction, the contents of the core register *t0* (the address for the canary) is sent through the *rs1[63:0]* field of the RoCCIO. The PUF is implemented as a SRAM memory initialized to random default values of 64-bit wide words. For our proof-of-concept implementation, we have used a PUF with 1024 unique challenge-response pairs. This was sufficient for our implementation since our RocketChip's program memory configuration was generated with a usable stack memory of less than 1024 bytes for the sake of simplicity. However, in practical applications, PUFs with larger CRPs are required since all 64 bits of the address or the optimized 32 bits will be used. When a `ce_fetch` instruction is interpreted the appropriate read control signals are sent to the PUF and the binary secret register. The address in *rs1[63:0]* is sent to the Challenge Generator, which prepares the PUF challenge by selecting the appropriate bits from the address and sends it to the PUF. The default random word in the memory-PUF for that particular address (challenge) serves as the response. It is to be noted that since this is a PUF, the random default values in the 1024 locations are static and is a signature of the PUF, and hence it will not change. The response value is read and sent to

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

the XOR gate. Simultaneously, the value in the binary secret register is also read and sent to the XOR gate. The output of the XOR gate is the resultant canary word and is fed to the *rd[63:0]* field of the response interface. The response is sent back to the core by writing the value in *rd[63:0]* to the register *t0* on the core, as indicated by the RoCC instruction.

G. Software Design with PUF Canaries

A program that needs to be protected with PUF-based canaries is compiled in the following fashion:

Step 1 – Generating assembly: We initially compile the program to an intermediate assembly code which allows us to scan the code to identify the target buffers to protect. This is demonstrated with the example C function in Fig. 8 and the corresponding assembly code in Fig. 9. Note that this function is for demonstration purposes only, it is not actually a vulnerable function. We analyze each stack frame and find the individual buffer locations based on the stack pointer or base pointer offset. We calculate the extra memory space required in the buffer to introduce the canaries, and the specific locations in the stack frame where those words will be placed. We also recalculate the references for variables and buffers in the stack.

Step 2 – Modification of assembly: The assembly code modification involves two major operations – canary placement and canary validation. The modified assembly code is shown in Fig. 10. For the canary placement, we first expand the stack frame by modifying the function prologue. In the example shown in Fig. 9, we will be placing 2 canaries, hence we subtract 2 x 8 bytes = 16 bytes to the stack pointer. Hence we replace `add sp,sp,-96` with `add sp,sp,-112`. We update all the references that use the stack pointer, such as the location where the return address is saved. For each buffer we place the canary in the following manner. First the address where the canary is to be placed is loaded on to the *t0* register. In our example, for the first buffer, we choose the canary location as -32(s0). This address indicates the address on top of the buffer. We use `add t0,s0,-32` to load the address onto *t0*. Now, we craft our `ce_fetch` custom instruction. A generic 32-bit RoCC instruction extends the RISC-V ISA and is encoded in the format as shown in Fig. 4(b). There are four RoCC instructions available (*custom0-3*) that are identified by the 7-bit opcode field, as shown in Table II. The *funct7* field can be used to further specify a particular function of the RoCC instruction. We use *custom0* to implement the canary instructions. We set the *funct7* field to b'0000000 (0) for `ce_fetch`. The *rs1* field is set to the *t0* register (b'00101), and the *rd* field is also set to *t0*. The corresponding *xs1* and *xd* fields are set to 1. The final crafted `ce_fetch` instruction is represented by `0x1714b`. The `ce_init`, `ce_set` and `ce_reset` instructions are also crafted similarly with *funct7* set to 1, 2 and 3 respectively (the details are omitted for brevity). We repeat the same process for the second buffer as shown in the example. Next, we readjust the location of the buffers by subtracting 8 and 16 bytes from the original addresses. This is accomplished by changing the buffer references for the `memcpy` function as `add a4,s0,-64` (for the first buffer). Immediately after the `memcpy` function returns, we place our canary validation code. This needs to be done for any copy to buffer function, such as `memcpy`, `strcpy`, etc. First we load the canary value on the stack onto the register *t1* by `lw t1,-32(s0)`. Next, we follow the same steps as before to fetch the actual canary value for that location from the Canary Engine into the register *t0*. We compare the values in registers *t0* and *t1* and proceed or halt depending on a match or mismatch. The same process is repeated for all the buffers on the stack. Finally, at the function epilog, we update the references for fetching the return address, and the stack and base pointers (Fig. 10). In the validation process, our design only scans for standard library buffer copy functions. However, it will not be able to detect manual writes to a buffer using a loop. Such cases may be handled using LLVM compiler toolchains where the copy operations can be parsed from intermediate representations (IR). The assembly modification requires identifying the number of buffers, their size and their offset from the symbol used to reference the buffers. The canary address for a buffer can be calculated using the size and offset for the buffer. The stack pointer needs to be calculated accordingly to make space for the canaries. The references to the buffers also need to be

```
1.  void func1()
2.  {
3.   int var1;
4.   char buffer1[32];
5.   int var2;
6.   char buffer2[32];
7.   var1 = 1;
8.   var2 = 2;
9.   memcpy(buffer1,"hello",5);
10.  memcpy(buffer2,"world",5);
11. }
```

Fig. 8. Example C function.

```
1.  func1:
2.      add     sp,sp,-96
3.      sd      ra,88(sp)
4.      sd      s0,80(sp)
5.      add     s0,sp,96
6.      li      a5,1
7.      sw      a5,-20(s0)
8.      li      a5,2
9.      sw      a5,-24(s0)
10.     add     a4,s0,-56
11.     li      a2,5
12.     lui     a5,%hi(.LC1)
13.     add     a1,a5,%lo(.LC1)
14.     mv      a0,a4
15.     call    memcpy
16.     add     a4,s0,-88
17.     li      a2,5
18.     lui     a5,%hi(.LC2)
19.     add     a1,a5,%lo(.LC2)
20.     mv      a0,a4
21.     call    memcpy
22.     nop
23.     ld      ra,88(sp)
24.     ld      s0,80(sp)
25.     add     sp,sp,96
26.     jr      ra
```

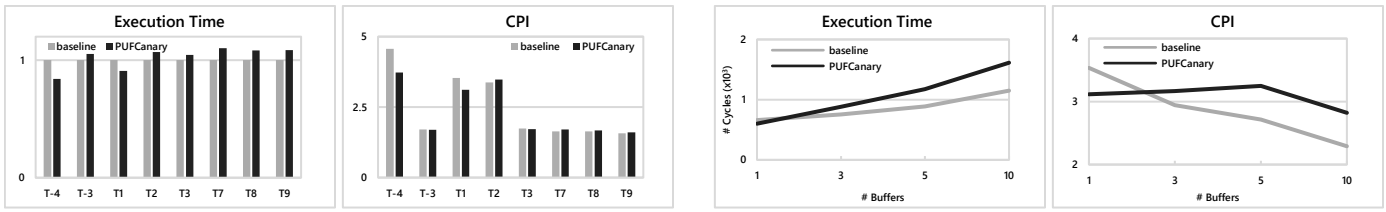
Fig. 9. Disassembled code.

```
1.  func1:
2.      add     sp,sp,-112
3.      sd      ra,104(sp)
4.      sd      s0,96(sp)
5.      add     s0,sp,112
6.  # Place canary @ -32(s0)
7.      add     t0,s0,-32
8.      .word   0x1714B
9.      sw      t0,-32(s0)
10. # Place canary @ -72(s0)
11.     add     t0,s0,-72
12.     .word   0x1714B
13.     sw      t0,-72(s0)
14.     li      a5,1
15.     sw      a5,-20(s0)
16.     li      a5,2
17.     sw      a5,-24(s0)
18.     add     a4,s0,-64
19.     li      a2,35
20.     lui     a5,%hi(.LC1)
21.     add     a1,a5,%lo(.LC1)
22.     mv      a0,a4
23.     call    memcpy
24. # Validate canary @ -32(s0)
25.     lw      t1,-32(s0)
26.     add     t0,s0,-32
27.     .word   0x1714B
28.     bne     t0,t1,_die
29.     add     a4,s0,-104
30.     li      a2,5
31.     lui     a5,%hi(.LC2)
32.     add     a1,a5,%lo(.LC2)
33.     mv      a0,a4
34.     call    memcpy
35. # Validate canary @ -72(s0)
36.     lw      t1,-72(s0)
37.     add     t0,s0,-72
38.     .word   0x1714B
39.     bne     t0,t1,_die
40.     nop
41.     ld      ra,104(sp)
42.     ld      s0,96(sp)
43.     add     sp,sp,112
44.     jr      ra
```

Fig. 10. The modified assembly code. Canary placement and validation code are shown in boxes.

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



(a) (b) (c) (d)

Fig. 11. Wilander benchmark evaluation w.r.t. (a) execution time (normalized), and, (b) effective CPI; Performance overhead trends for multiple buffer protection w.r.t (c) execution time (cycles), and (d) CPI.

calculated considering the extra space for the canaries. For the canary validation, the symbol referencing the buffer can be used to identify the location of the canary. A table can be maintained with the IR to match the buffer to its canary location.

Step 3 – Final Compilation: The final PUFCanary enforced assembly code is passed to the compiler to assemble, link and generate the final executable binary of the program. No compiler modifications are necessary to embed the instructions in the final binary since we provided the custom instruction as a binary instruction word, and the RoCC instruction format is already supported by the RISC-V GNU toolchain. Clang/LLVM can be used to automate the entire process. The automation involves emitting the LLVM IR from the source, writing compiler passes for the IR to modify the assembly by adding canary placement and validation codes, and compiling the IR after the passes into machine code using Clang.

H. Experimental Results

We tested our Canary Engine in the C++ cycle accurate emulator of the RocketChip Generator, and on the Xilinx Zybo FPGA. The hardware architecture of the Canary Engine is coded in CHISEL and is translated to synthesizable Verilog code using the available tools in the RocketChip Generator. We evaluated the security of our design and the performance overheads using the Wilander Buffer Overrun Suite [46]. We compiled the tests using the RISCV GNU GCC compiler in two versions: (i) the *baseline* code without canary protections, and (ii) *PUFCanary* code with our canary protection. Since our proof-of-concept design only protects the stack, we considered only the 12 stack-based test cases in the test suite. However, due to the limitations in the RISC-V toolchain, we were able to port only 8 test cases. The 4 test cases (with LONGJMP) could not be ported. For the 8 working test cases (shown in Table III), our *PUFCanary* was able to prevent all 8 attack cases.

Fig. 11(a-b) shows the performance overheads of *PUFCanary* over the *baseline* code. The corresponding instruction overheads are shown in Table IV. The execution time overhead of *PUFCanary* over *baseline* is 2.3% on average across the 8 test cases. There is none or negligible effect on CPI (cycles per instruction), where our tests reveal 0.97X average overhead. Few of the test cases show improvement in execution time; this may be due to architectural optimizations such as better cache performance. Our results are better than the original StackGuard [1] which shows 6% overhead in the best case. Furthermore, our results are comparable to HDFI [31] which also has ~2% overhead. To further support our claim of multiple buffer protection performance, we modified the test case T1 to include multiple buffers. We enforced canary protection on 1, 3, 5, and 10 buffers in the same vulnerable stack of T1. The performance trends in case of multiple buffer protection are shown in Fig. 11(c-d). The execution time overhead trend is mostly linear with 10-buffer protection having 1.5X the overhead of 1-buffer protection. Similar trend can be observed with CPI overhead where 10-buffer protection overhead is 1.4X that of 1-buffer protection. This shows that a fine-grained protection with our Canary Engine does not have a significant performance impact. The Canary Engine RoCC module with a 1024 CRP PUF exhibited ~2.9% area overhead over a vanilla RocketChip with the default configuration.

TABLE III. Wilander Test Cases for Stack Corruption

Case	Description
T-4	Overflow all the way to FUNCTION PTR as PARAM
T-2	Overflow of a PTR, then pointing at FUNCTION PTR as PARAM
T1	Overflow all the way to RETURN ADDRESS
T2	Overflow all the way to OLD BASE POINTER
T3	Overflow all the way to FUNCTION PTR as local variable
T7	Overwrite of a PTR to point at RETURN ADDRESS
T8	Overwrite of a PTR to point at BASE POINTER
T9	Overwrite of a PTR to point at FUNCTION PTR as variable

TABLE IV. Instruction Overheads

Case	Overhead %
T-4	3.00
T-2	5.98
T1	3.21
T2	3.57
T3	5.88
T7	6.05
T8	6.26
T9	6.07

V. FIXER Security Architecture

A. FIXER Design for Backward-Edge CFI

C programs compiled with the GNU GCC Toolchain for RISC-V target architecture do not provide any protection against memory corruption vulnerabilities such as, buffer overflow. An adversary can provide malicious inputs to a program and can overwrite the return address of a function and redirecting the control flow of the program. In FIXER, we implement the Shadow Stack security primitive to enforce CFI at the backward edge (return to functions). The RoCC is used to implement the Shadow Stack, thus preventing the need to modify the core system architecture. The Shadow Stack is designed as a hardware memory on the RoCC. Fig. 12 shows the steps for detecting CFI violation using a Shadow Stack. The return address is pushed on the system stack by default when a function call is made in the program. During this time, same

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

return address is sent using a RoCC custom instruction to the RoCC to push it on the Shadow Stack as a backup. The return address is popped from the system stack to the instruction pointer register for execution when returning from a function. During this return the RoCC Shadow Stack is queried to retrieve the backup return address and compare against the one from the system stack. If they match, the program proceeds with normal execution, else a potential memory corruption is detected, and program execution is stopped. Note that compared to HAFIX [27] where Shadow Stack is part of core, FIXER implements it in the coprocessor leaving the core architecture untouched. It is to be noted that FIXER is complementary to existing DEP protection, since the FIXER instructions must be tamperproof to ensure protection.

Fig. 13(a) details the software design flow for FIXER. The source code is first marked with CFI tags (for saving to shadow stack and validation) and compiled to an intermediate assembly code using the RISC-V GNU toolchain. The assembly code is parsed by expanding the tags and injecting the required RoCC instructions in the assembly. The lifted assembly code is generated using a custom parsing script or a compiler pass and then assembled and linked to produce the fully compiled RISC-V binary. These steps are further elaborated in Section V.B.

Fig. 13(b) shows the hardware design flow for FIXER (coded in CHISEL [36] as a RoCC). The hardware implementation of FIXER in RoCC is described in Section V.C. The RocketChip with the RoCC is then compiled with the generator to output the synthesizable Verilog code and the FPGA bitstream. The RISC-V Linux system image, the FPGA devicetree and the generated bitstream are then deployed to the FPGA to run RocketChip. This FIXER assisted RocketChip system can successfully protect against CFI violations on the RISC-V programs compiled with FIXER assisted compilation process.

B. RISC-V Software Design with FIXER

Any program that needs to be backward-edge CFI enforced, is compiled and processed by the following steps:

Step 1 - Source code annotation: We annotate the function calls and returns with a special tag to indicate the sites where the enforcement needs to take place. We use `CFI_CALL` tag before a function call and a corresponding `CFI_RET` tag just before a return from the called function, as shown in Fig. 14.

Step 2 – Tag expansion: We expand the CFI tags to actual RISC-V assembly instructions. During compilation, we intercept the intermediate assembly code of the program and inject the RoCC custom instructions to communicate with the RoCC. Fig. 15 shows the assembly instructions corresponding to `CFI_CALL` and `CFI_RET`, that are placed just before the `call` and `jr ra` (return) instructions respectively.

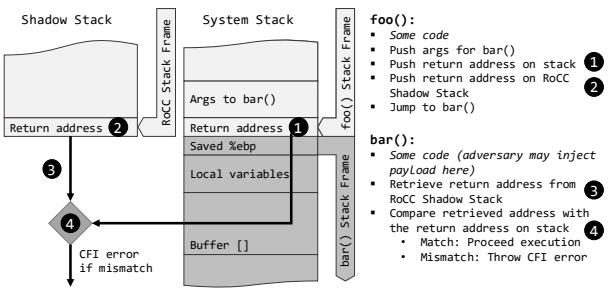


Fig. 12. CFI violation detection using a Shadow Stack.

For `CFI_CALL`, we first retrieve the current value of the program counter from the instruction pointer register using the `auipc` instruction and add 14 bytes offset (instructions are variable length) to calculate the target return address. We save the computed return address in a temporary register *t0*. Then we craft the RoCC instruction `cfi_call` to push the return address from *t0* to the Shadow Stack. A generic 32-bit RoCC instruction extends the RISC-V ISA and is encoded in the format as shown in Fig. 4(b). There are four RoCC instructions available (*custom0-3*) that are identified by the 7-bit opcode field, as shown in Table II. The *funct7* field can be used to further specify a particular function of the RoCC instruction. We use *custom0* to implement the CFI instructions. We set the *funct7* field to b'0000000 (0) for `cfi_call` and to b'0000001 (1) for `cfi_ret`. We use the *rs1* field to set it to use the t0 register (b'00101), where we temporarily stored the computed return address and set the corresponding *xs1* bit to 1. The final crafted `cfi_call` instruction is represented by 0x0002a00b.

For `CFI_RET`, we set the *funct7* field to b'0000001 (1) and set the *rd* field to use the *t0* temporary register (b'00101) along with *xd* bit as 1. The final crafted custom instruction word for `cfi_ret` is represented by 0x0200428b. During a return from a function, the saved return address is popped from the system stack on to the link register *ra*. We then use the `cfi_ret` to retrieve the backup return address from the RoCC Shadow Shack on to register *t0*. The value in *t0* is then compared against the value in the register *ra* using the `bne` instruction. If they match, the execution proceeds by completing the return (`jr ra`: jump register), else we throw a CFI error.

Step 3 – Compilation: The final CFI enforced assembly code is passed to the compiler to assemble, link and generate the final executable binary of the program. No compiler modifications are necessary to embed the instructions in the final binary since we provided the custom instruction as a binary word, and the RoCC instruction format is supported by the GNU toolchain.

C. FIXER Hardware Implementation in RoCC

Fig. 16 shows the FIXER implementation in the RoCC. The program binary runs on the Rocket Core and sends RoCC instructions over the RoCCIO whenever a security validation is required. The RoCC instruction is first passed through the Cmd decoder, which extracts the individual fields of the RoCC instruction, and the contents of the two registers *rs1* and *rs2* if specified. The opcode field is decoded to the *custom0* instruction in our implementation. The *funct7* field is decoded to interpret a `cfi_call` or a `cfi_ret`.

For `cfi_call`, the contents of core register *t0* (the return address) is sent through the *rs1[63:0]* field of the RoCCIO interface. The shadow stack is implemented as a SRAM memory with 64-bit wide words. A top-of-stack register (ToS)

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

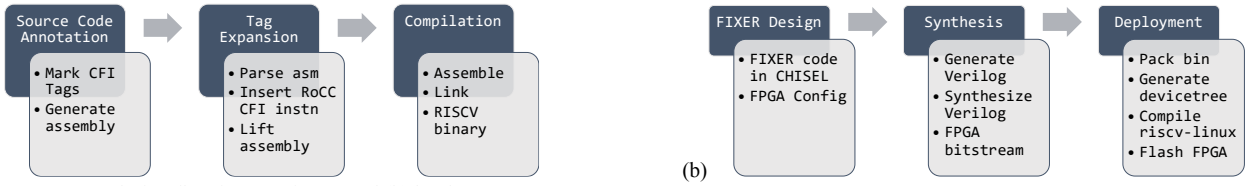


Fig. 13. FIXER design flow in (a) software and (b) hardware.

holds the address of the top of the shadow stack. If a `cfi_call` is interpreted, the content of the ToS register is incremented by 1. The updated value in the ToS register is used to decode the write address for the shadow stack. The value in the *rs1* field is written to this address on the shadow stack. This operation is non-blocking, so the core can continue execution after issuing the `cfi_call` instruction. There is a command queue at the RoCCIO interface to prevent race conditions. If the instruction function is interpreted as `cfi_ret`, then the ToS register is read to obtain the address for the shadow stack. This address is used to read the saved return address from the shadow stack memory. The value is then sent back to the core by writing to the *rd[63:0]* field of the response interface of the RoCCIO, which writes the value to the *t0* register on the core as indicated by the RoCC instruction. Our proof-of-concept implementation of the shadow stack can accommodate 1000 addresses. However, this can be updated on demand by reconfiguring the FIXER module on the FPGA, a benefit exclusive to our implementation.

D. *Forward-edge Protection with FIXER*

A shadow stack only protects control flow on return boundaries. However, programs often use function pointers to jump to multiple function addresses. To ensure the validity of such function calls using function pointers, a pre-computed call policy is enforced. A static or runtime analysis is performed on the program to construct a control flow graph (CFG), represented as a policy matrix that indicates the valid call targets for each function call made using a function pointer. The policy matrix is loaded in memory and at runtime, it is queried to validate the call target for every indirect function call. This forward-edge protection is implemented as another FIXER security module (Fig. 16). The policy matrix memory is created in RoCC along with caller and callee address decoders. Our proof-of-concept implementation has 64 rows (each represents an originating call site address) in the matrix and each row holds a 64-bit policy vector (each bit represents a call target address). A set (unset) bit indicates that the call is valid (invalid) for that (caller, callee) pair. A RoCC instruction `cfi_matld` is used to load the policy bitmap into the FIXER module prior to the program execution. A RoCC instruction `cfi_fwd` is inserted before every indirect function call in the source code. The `cfi_fwd` instruction sends the caller and the dereferenced function pointer (callee) addresses to the RoCC for validation. The forward-edge FIXER module validates the action using the policy matrix and sends back a 1 (0) to allow (disallow). Similar to the shadow stack implementation, the policy matrix size can also be updated post-deployment by reconfiguring the FPGA.

E. *Security Implications and Benefits*

FIXER is targeted for hybrid architectures, e.g., CPU+FPGA, or ASIC+FPGA. Our current results are based on both the RocketChip and the RoCC accelerator being on the FPGA since we do not have access to such architecture. It is true that if the FPGA is off-chip, there could be performance degradation (due to speed gap between CPU and FPGA) if the checking is performed in a synchronous and fine-grained manner. Performance issues can be alleviated by making the checking asynchronous using interrupts. In such cases, the program can continue execution, until the FPGA raises an interrupt to halt the program. However, it cannot be guaranteed that the adversary has not been able to take control of the system before the FPGA detects the attack. When the FPGA is on-chip, e.g., Intel Xeon with embedded FPGA, performance overheads can be alleviated due to QuickPath Interconnect (QPI) interface between the core and the FPGA for fast communication.

FIXER implemented on the FPGA offers benefits compared to other core based or system level protection schemes. Designs e.g., NILE which use the virtual address space to house the shadow stack cannot scale based on the branch sequence depth. HAFIX has a separate limited memory on the core to store the CFI tags. However, in case of FIXER, the design can be scaled

```
void main () {
    ...
    CFI_CALL
    myFunc();
    ...
}
```

```
void myFunc() {
    ...
    CFI_RET
    return;
}
```

Fig. 14. Source code annotation

```
# CFI_CALL
auipc   t0,0
add     t0,t0,14
.word   0x0002a00b
call    myFunc
```

```
# CFI_RET
.word   0x0200428b
 bne     t0,ra,_cfi_error
jr      ra
```

Fig. 15. Tag expansion

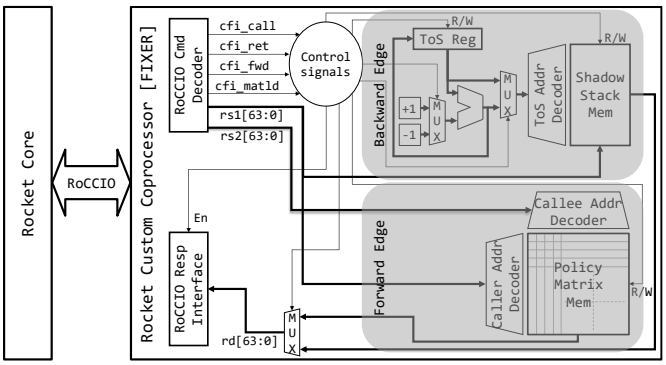


Fig. 16. FIXER implementation in RoCC.

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

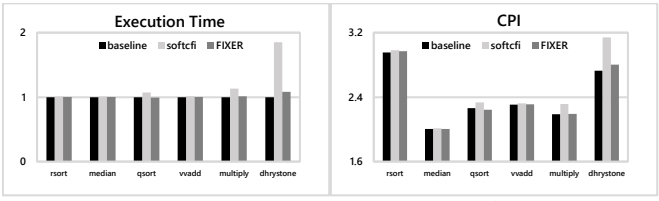
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

up or down based on the actual workload of the system. Typically, embedded devices e.g., IoTs have a limited set of workloads, and FIXER module on the IoT's SoC can be scaled appropriately based on the workload. For example, if a new workload introduced to the system requires a larger shadow stack, the FPGA can be reconfigured to accommodate that (the maximum size being limited by available LUTs).

F. Experimental Results

The hardware architecture of FIXER is coded in CHISEL and translated to synthesizable Verilog using the available tools in the RocketChip generator. We prepared a FPGA system image using the generated Verilog and ran it on a Xilinx Zynq FPGA. A sample program is written with 1 billion iterations of function calls and returns. One version of the code implemented a simple software version of the shadow stack (*softcfi*). The software shadow stack is created as a regular stack in the address space. During function calls, the return address is simultaneously placed on the system stack as well as the shadow stack. Another version instrumented the code with the RoCC CFI instructions (*FIXER*). We compiled the *baseline* (no CFI checks), the *softcfi* and *FIXER* versions using the RISC-V GNU toolchain. The three versions of the program were run on the system running on the FPGA. The base code takes 19 seconds to execute, whereas the *softcfi* takes 74 seconds. *FIXER* takes 29 seconds resulting in ~1.5X overhead over *baseline* and ~2.55X lower overhead compared to *softcfi*. The FPGA on idle draws 370mA current, while on load (with the program running) draws 420mA current, resulting in 1.13X increase. The corresponding energy overhead is 3.89X for *softcfi* and only 1.53X for the *FIXER* (60.52% improvement). The FIXER RoCC module incurs only 2.9% area overhead over the vanilla RocketChip without RoCC.

We evaluated FIXER performance by enforcing it on RISC-V architecture benchmarks. The benchmarks are modified to create three versions for comparison: (i) *baseline* with no CFI enforcement, (ii) *softcfi* with the software-based CFI enforcement, and (iii) *FIXER* with RoCC based CFI protection. We ensured that the benchmark code remains the same across all the three versions except the CFI enforcement code. We compiled the benchmarks with the RISC-V GNU toolchain without compiler optimizations and ran the compiled binaries on the Zynq FPGA. Fig. 17 show the evaluation results for backward-edge FIXER. The instruction overheads are shown in Table V. With the backward-edge protection, the execution time overhead with *softcfi* is ~18% on average across the six benchmarks compared to 1.5% with *FIXER*. The *softcfi* increases the CPI (cycles per instruction) by 4.6% over the *baseline*, while the *FIXER* increases the CPI by only 0.5%. With the forward-edge protection, the execution time overhead with *softcfi* is ~2% on average across the six benchmarks compared to 0.61% with *FIXER* and CPI reduces 0.4% on average, which is negligible.



(a) (b)

Fig. 17. RISC-V benchmark evaluation for backward-edge protection w.r.t. (a) execution time (number of cycles), and (b) effective CPI.

TABLE V. Benchmark Instruction Overheads

	Backward-edge	Improvement over *softcfi*
rsort	1.000019X	0.0126%
median	1.000305X	0.2310%
qsort	1.00434X	3.1770%
vvadd	1.000622X	0.5080%
multiply	1.008037X	5.7140%
dhrystone	1.068607X	32.7930%

VI. Limitations and Opportunities

A. PUFCanary

PUF design decisions: We have implemented a simplified version of PUF. The security of the design is dependent on the number of CRPs that the PUF can generate. A smaller PUF with limited CRPs can lead to duplicate canaries, potentially allowing the attacker to guess the canary for different addresses. To get around this security limitation, the output of the XOR gate can be combined with the original address and hashed to generate a 64-bit canary. For higher security, a larger SRAM PUF can be used for a 1:1 address-to-canary mapping at the cost of area and power overheads. To optimize the overhead, we can down-size the SRAM footprint, however, this can cause read disturb failures during query. It has been shown in literature that PUFs can be used to reliably generate random numbers by targeted NBTI aging [47]. If raw SRAM PUF responses are not uniformly random, PUF responses can be transformed to high-entropy random values by fuzzy extraction [51]. This may add to the performance overhead if the PUF is queried frequently. Cryptographic engines in processors often provide hardware PUFs and TRNGs which can be reused in the Canary Engine. Our proposed technique is not limited by the choice of the PUF used, and is also applicable to other PUF flavors e.g., arbiter PUF [37], flip-flop PUF [39], etc. MRAM/STTRAM PUFs [49-50] that guarantee uniform randomness may also be used to eliminate the need for post-processing. In our SRAM PUF, we have a synchronous interface between the core and the Canary Engine, which contributes to some of its performance overhead. In practical implementations, the Canary Engine does not need to be stateless and can be pipelined to improve performance.

Canary validation decisions: In our design, we have validated all the canaries in a function stack frame after every write to a buffer. Although more complex and performance intensive, it can detect data-oriented attacks proactively. However, the canaries can also be validated all at once in the function epilogue to reduce design complexity at the cost of attack detection when the function returns. It is possible that a non-control data attack may succeed before detection in this case. Control-flow bending attacks using a buffer overflow

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

vulnerability can only be detected if the control-data is used after the validation of the corrupted canary takes place. If the canaries are validated after the control data is used to bend the control flow, the attack may still succeed.

Heap/bss protection: In this work, we have targeted the protection of stack only, however the design can be extended for heap and bss protection by expanding the PUF challenge set for the larger address space.

Buffer overread protection: PUFCanary can only detect buffer overwrites and not overreads. This leaves the system open to memory disclosure and can potentially leak the canaries. However, since all the canaries are random and unique, disclosure of one canary may not provide the adversary enough opportunities to launch a control-flow or data-flow attack unless there is a buffer-overflow and a buffer-overread vulnerability on the same or nearby buffers. This is possible, for example, in a loop which contains a buffer vulnerable to both overread and overwrite. If, due to a memory disclosure attack at that location, the adversary learns the canary, he can reuse that canary for the same location for the overflow attack. Aside from this scenario, the adversary cannot reuse the same canary found from disclosure to attack a different buffer, since they are protected by different canary words. However, in case of a smaller PUF design with repeated canaries, it may be easier for an adversary to reuse canaries in case of memory disclosures.

Data-oriented attacks: PUFCanary can detect data-oriented attacks that originate from a buffer overflow vulnerability. However, other data-oriented attacks that originate from memory disclosures, format-string vulnerabilities or integer overflows cannot be detected by PUFCanary.

B. FIXER

Multi-process protection: Our implementation of FIXER enforces protection for a single process only. For a simultaneous multi-process protection, the FIXER design can be expanded to accommodate multiple shadow stacks and policy memories for different processes. A round-robin scheduler on the FIXER module can assign the shadow stacks and policy memories to each process based on the process ID.

Tamper protection: The FIXER module on the FPGA also needs to be protected from tampering or data leaks. The current RocketChip implementation allows the entire code containing custom RoCC instructions to be run with supervisor privileges. This can be restricted via system calls so that RoCC instructions are first verified and then run with supervisor privileges.

Buffer overread protection: It should be noted that FIXER is still vulnerable to buffer over-reads. Similar to HAFIX and NILE, FIXER can not enforce security if the adversary can modify binary to skip the custom instructions.

VII. Conclusion

We presented randomized stack canaries for fine grained buffer overflow detection. Our unique PUF based approach allows multiple canaries to be placed in the stack frame, providing a lightweight, yet secure way of detecting buffer overflow vulnerabilities. We also presented FIXER, a more performance-friendly low-power reconfigurable CFI security architecture to implement a shadow stack and a policy memory in a RISC-V coprocessor for uninterrupted program flow. FIXER provides fast and efficient CFI checking whereas PUFCanary provides better protection against overflow vulnerabilities at the cost of design complexity and slight performance loss. Simulation results using RocketChip show the effectiveness of our approach.

References

[1] Abadi et al. "Control-flow integrity." In Proc. ACM CCS, 2005.

[2] Team, PaX. "PaX address space layout randomization (ASLR), 2003." URL: https://pax. grsecurity. net/docs/aslr.txt

[3] Data Execution Prevention, https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx

[4] Cowan et al. "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks." In SSYM, 1998.

[5] Castro et al. "Securing software by enforcing data-flow integrity." OSDI, 2006.

[6] Dhawan et al. "Architectural support for software-defined metadata processing." SIGARCH Computer Arch News, 2015.

[7] Song *et al*., "HDFI: Hardware-Assisted Data-Flow Isolation," *IEEE Symposium on Security and Privacy (SP), 2016.*

[8] R. Pappu, "Physical one-way functions," PhD thesis, Massachusetts Institute of Technology, 2001.

[9] Asanovic et. al, "The Rocket Chip Generator", technical report, www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[10] www.extremetech.com/extreme/184828-intel-unveils-new-xeon-chip-with-integrated-fpga-touts-20x-performance-boost

[11] Delshadtehrani et al. "Nile: A Programmable Monitoring Coprocessor," in *IEEE Computer Architecture Letters*, 2018.

[12] Park et al. "Microarchitectural Protection Against Stack-Based Buffer Overflow Attacks." *IEEE Micro*, 2006.

[13] Nishiyama et al. "SecureC: control-flow protection against general buffer overflow attack," C*OMPSAC*, 2005.

[14] Sinnadurai et al. "Transparent runtime shadow stack: Protection against malicious return address modifications," 2008.

[15] Zeitouni et al. "ATRIUM: runtime attestation resilient under memory attacks." ICCAD 2017.

[16] Iwainsky et al. "Compiler Supported Sampling through Minimalistic Instrumentation," ICPPW, 2014.

[17] Pappas et al. "Transparent ROP exploit mitigation using indirect branch tracing." In USENIX SEC, 2013.

[18] Cheng et al. "ROPecker: A Generic and Practical Approach For Defending Against ROP Attack." NDSS Symposium 2014.

[19] Alves et al. "TrustZone: Integrated hardware and software security." ARM white paper, 2004.

[20] McKeen et al. "Innovative instructions and software model for isolated execution." In HASP@ ISCA, 2013.

[21] Intel: Control-Flow Enforcement Technology Review, 2016.

[22] Ramakesavan et al. "Intel memory protection extensions (intel mpx) enabling guide," 2015.

[23] Yoo et al. "Performance evaluation of Intel® transactional synchronization extensions for high-performance computing." SC-Intl Conf for HPC, Networking, Storage and Analysis. 2013.

[24] Kasikci et al. "Failure sketching: a technique for automated root cause diagnosis of in-production failures." In SOSP, 2015.

[25] Dhawan et al. "Architectural support for software-defined metadata processing." SIGARCH Computer Arch News, 2015.

[26] Wang et al. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity." In IEEE S&P, 2010.

[27] Davi et al. "HAFIX: hardware-assisted flow integrity extension." in DAC, 2015.

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2020.2984407, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

[28] Jin et al. "Hardware control flow integrity." *The Continuing Arms Race*. ACM and Morgan & Claypool, 2018.

[29] Ge et al. "GRIFFIN: Guarding Control Flows Using Intel Processor Trace" In ASPLOS, 2017.

[30] Arias et al. "HA^{2}lloc: Hardware-Assisted Secure Allocator." in HASP, 2017.

[31] Song *et al*., "HDFI: Hardware-Assisted Data-Flow Isolation," *IEEE Symposium on Security and Privacy (SP), 2016.*

[32] Bresch et al. "A red team blue team approach towards a secure processor design with hardware shadow stack," *IVSW*, 2017.

[33] Bresch et al. "Stack Redundancy to Thwart Return Oriented Programming in Embedded Systems," in *IEEE ESL*, 2018.

[34] Panis et al. "Scaleable shadow stack for a configurable DSP concept," in *IWSOC, 2003.*

[35] Ming et al. "Shadow Stack Scratch-Pad-Memory for Low Power SoC," in *IEEE Intl Symposium on Embedded Computing*, 2008.

[36] Bachrach *et al*., "Chisel: Constructing hardware in a Scala embedded language," In *DAC, 2012*.

[37] Lim et al, "Extracting secret keys from integrated circuits." IEEE Trans. on VLSI Sys., vol 13, no. 10 (2005).

[38] G. E. Suh, S. Devadas, Physical unclonable functions for device authentication and secret key generation," DAC, 2007.

[39] Zheng et al, "ScanPUF: robust ultralow-overhead PUF using scan chain," ASP-DAC, 2013.

[40] Petrie et al, "A noise-based IC RNG for applications in cryptography," IEEE Trans. Circuits Syst. I, vol. 47, no. 5, pp. 615–621, May 2000.

[41] Sunar et al, "A provably secure TRNG with built-in tolerance to active attacks," IEEE Trans. Comput., vol. 56, no. 1, pp. 109–119, Jan. 2007.

[42] Brederlow et al, "A low-power TRNG using random telegraph noise of single oxide-traps," in IEEE ISSCC Dig. Tech. Papers, 2006.

[43] Schellekens et al, "FPGA vendor agnostic TRNG," in Proc. 16th Int. IEEE Conf. Field Programmable Logic and Applications, 2006.

[44] Kinniment et al, "Design of an on-chip random number generator using metastability," in Proc. ESSCIRC, 2002, pp. 595–598.

[45] Yasuda et al, "Physical RNG based on MOS structure after soft breakdown," IEEE J. Solid-State Circuits, vol. 39, no. 8, 2004.

[46] Wilander et al, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention." in *NDSS*. Vol. 3. 2003.

[47] Mathew *et al*., "16.2 A 0.19pJ/b PVT-variation-tolerant hybrid physically unclonable function circuit for 100% stable secure key generation in 22nm CMOS," *in ISSCC*, 2014.

[48] Nyman et al. "HardScope: Hardening Embedded Systems Against Data-Oriented Attacks", in DAC 2019.

[49] X. Zhang *et al*., "A novel PUF based on cell error rate distribution of STT-RAM," *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Macau, 2016, pp. 342-347.

[50] Das et al., "MRAM PUF: A Novel Geometry Based Magnetic PUF With Integrated CMOS," in *IEEE Transactions on Nanotechnology*, vol. 14, no. 3, pp. 436-443, May 2015.

[51] Suzuki et al., "Efficient Fuzzy Extractors Based on Ternary Debiasing Method for Biased Physically Unclonable Functions," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 66, no. 2, pp. 616-629, Feb. 2019.

0278-0070 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Penn State University. Downloaded on June 22,2020 at 19:42:38 UTC from IEEE Xplore. Restrictions apply.