SCRAMBLE-CFI: Mitigating Fault-Induced Control-Flow Attacks on OpenTitan

Pascal Nasahl
Graz University of Technology
Graz, Austria
pascal.nasahl@iaik.tugraz.at

Stefan Mangard
Graz University of Technology
Graz, Austria
stefan.mangard@iaik.tugraz.at

ABSTRACT

Secure elements physically exposed to adversaries are frequently targeted by fault attacks. These attacks can be utilized to hijack the control-flow of software allowing the attacker to bypass security measures, extract sensitive data, or gain full code execution.

In this paper, we systematically analyze the threat vector of fault-induced control-flow manipulations on the open-source OpenTitan secure element. Our thorough analysis reveals that current countermeasures of this chip either induce large area overheads or still cannot prevent the attacker from exploiting the identified threats. In this context, we introduce SCRAMBLE-CFI, an encryption-based control-flow integrity scheme utilizing existing hardware features of OpenTitan. SCRAMBLE-CFI confines, with minimal hardware overhead, the impact of fault-induced control-flow attacks by encrypting each function with a different encryption tweak at load-time. At runtime, code only can be successfully decrypted when the correct decryption tweak is active. We open-source our hardware changes and release our LLVM toolchain automatically protecting programs. Our analysis shows that SCRAMBLE-CFI complementarily enhances security guarantees of OpenTitan with a negligible hardware overhead of less than 3.97 % and a runtime overhead of 7.02 % for the Embench-IoT benchmarks.

CCS CONCEPTS

• **Security and privacy → Hardware attacks and countermeasures**.

KEYWORDS

secure element, fault attacks, cryptographic control-flow integrity

ACM Reference Format:
Pascal Nasahl and Stefan Mangard. 2023. SCRAMBLE-CFI: Mitigating Fault-Induced Control-Flow Attacks on OpenTitan. In *Proceedings of the Great Lakes Symposium on VLSI 2023 (GLSVLSI '23), June 5–7, 2023, Knoxville, TN, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3583781.3590221

1 INTRODUCTION

In a fault attack, the adversary injects one or multiple bit errors into a chip by using non-invasive methods, such as voltage or clock glitching, or semi-invasive techniques that include shooting with a laser into the silicon of a decapsulated chip [1]. The effects of these bit errors can be exploited and enable the attacker to manipulate the control-flow of software executed on the system [6, 10, 16, 17].

Secure elements, such as OpenTitan [5], are used in smartphones and computers as a secure root of trust. As these chips run security-critical programs, such as a key storage and authentication services, they are lucrative fault targets. In addition, these devices are easily accessible for fault attackers as they are typically deployed in the wild. Hence, secure elements need to provide dedicated hardware- and software-based countermeasures protecting the execution of software against faults.

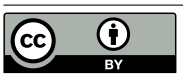
Contribution

In this paper, we first identify threat vectors enabling the adversary to hijack the control-flow of software by inducing faults into OpenTitan. Subsequently, we thoroughly analyze existing countermeasures aiming to mitigate these attacks.

Our analysis shows that existing countermeasures either induce large area overheads or inadequately reduce the attack surface to address control-flow manipulations. SCRAMBLE-CFI significantly confines the effect of these attacks with a minimal hardware overhead by introducing a cryptographically enforced control-flow integrity scheme. In SCRAMBLE-CFI, each function is encrypted with a different encryption tweak before the execution of the program. During runtime, the instrumented program automatically updates the decryption tweak in a CPU register. Only when the current execution context and the decryption tweak in the register match, the code can be successfully decrypted and executed. On control-flow redirections to functions outside of the call graph, the tweak mismatches and garbled instructions are decoded, triggering an alert. As SCRAMBLE-CFI utilizes the already existing scrambling unit of OpenTitan for the encryption, our countermeasure only requires minimal hardware changes, yielding an area overhead of less than 3.97 %. Furthermore, our performance analysis shows a small runtime overhead of 7.02 % for the Embench-IoT benchmarks.

In summary, our contributions are:

- We provide a systematic analysis of threat vectors on OpenTitan allowing an adversary to perform fault-induced control-flow manipulations.
- We discuss existing hardware- and software-based fault countermeasures integrated into OpenTitan.
- We introduce and open-source[1] SCRAMBLE-CFI, an encryption-based control-flow integrity scheme utilizing hardware features of OpenTitan.



GLSVLSI '23, June 5–7, 2023, Knoxville, TN, USA

ACM ISBN 979-8-4007-0125-2/23/06.
https://doi.org/10.1145/3583781.3590221

[1] https://extgit.iaik.tugraz.at/sesys/otcfi

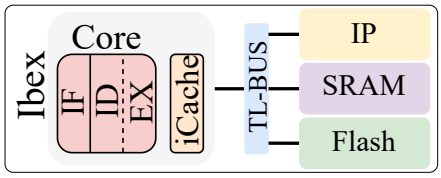


Figure 1: High-level overview of the OpenTitan chip.

- Finally, we discuss how SCRAMBLE-CFI, complementarily to existing countermeasures, enhances the resilience of OpenTitan against faults with a small runtime and area overhead.

2 BACKGROUND

This section provides background on the OpenTitan chip and control-flow integrity.

2.1 OpenTitan

OpenTitan is a secure element developed by Google and lowRISC and the entire design, including silicon as well as the firmware, is open-source. The goal of the project is to develop a chip that acts as a secure root of trust in different computing systems. OpenTitan contains a rich set of hardware- and software IP, including a key storage and an AES accelerator.

Figure 1 highlights the main architectural building blocks of the OpenTitan secure element. The Ibex 32-bit RISC-V processor is connected over the TileLink bus to the program memory (flash), the data memory (SRAM), and several other IP blocks. To provide data confidentiality, external data stored in flash or SRAM is encrypted by the OpenTitan scrambling unit. This engine encrypts all data using a round-reduced version of the PRINCE [2] cipher.

2.2 Control-Flow Integrity

Control-flow integrity schemes aim to detect fault-induced control-flow deviations from the intended control-flow. Here, these schemes [11, 14, 15] assign certain points in the program a unique control-flow signature during compile-time. At runtime, instrumented programs automatically derive the control-flow signature and compare the derived to the predefined signature. On a mismatch, a control-flow manipulation is detected. However, as the signature comparison induces performance overhead, the signature checks are only conducted at a coarse granularity, e.g., at the function or program end. Hence, an adversary might still be able to execute security-sensitive code before the control-flow manipulation is detected by CFI.

3 THREAT MODEL

Our threat model comprises an attacker with physical access to the OpenTitan chip performing fault attacks. This attacker is capable of injecting single or multiple faults into the secure element by using clock, voltage, or EM glitching techniques or by performing laser fault injection [1]. We assume that these faults cause single or multiple bit-flips [18] in the system. The goal of the attacker is to redirect the control-flow of software executed on the chip.

4 ANALYSIS

In this section, we discuss potential attack vectors within the presumed threat model (cf. Section 3) and systematically analyze existing hardware- and software-based countermeasures of OpenTitan aiming to address the identified threats.

4.1 Attack Vectors

The control-flow of software can be redirected at different control-flow manipulation (CFM) granularities:
CFM1: The attacker redirects the control-flow to a function that cannot be reached from the current execution context, i.e., escaping the call graph of the program.
CFM2: The control-flow is redirected from one branch target to the other when manipulating conditional branches.
CFM3: The attacker arbitrarily hijacks the control-flow within a function, i.e., to any basic-block.

The attack surface (A) comprises all elements of OpenTitan (cf. Figure 1). More specifically, a fault can be injected into CPU internal registers (**AR**), the instruction cache (**AIC**), the bus infrastructure (**AB**), or the memory (**AM**). Furthermore, the attacker can also target the core (**AC**), i.e., the instruction fetch, decode, and execute pipeline stages. For example, when targeting **AC**, the attacker can change the behavior of executed instructions when inducing bit flips into the instruction decoder or influence the comparison for a conditional branch in the ALU.

To manipulate the control-flow, we define three potential data targets (DT):
DT1: *Control-flow* related data comprises relative (DT1.1) and absolute (DT1.2) addresses as well as the program counter (DT1.3). To flip bits in relative addresses (DT1.1) used by unconditional and conditional branches, the attacker can inject faults into the immediate field of instructions stored in the program memory (**AM**) or the instruction cache (**AIC**) or transferred by the instruction bus (**AB**) [6]. Indirect calls can be manipulated by targeting absolute addresses (DT1.2a) stored in registers (**AR**) or the data memory (**AM**) or transferred by the data bus (**AR**). Moreover, returns can be redirected by flipping bits in return addresses (DT1.2b). Finally, the attacker also can directly inject faults into the program counter (DT1.3)(**AR**). Targeting **DT1** enables the adversary to arbitrarily manipulate the control-flow, i.e., **CFM1**, **CFM2**, and **CFM3**.
DT2: *Non-control-flow* related data, i.e., general purpose data, can be faulted by targeting the SRAM (**AM**) or the data bus (**AB**). When the faulted data is used by conditional branches [17], the attacker can influence their execution (**CFM2**).
DT3: *Instructions*, stored in the flash (**AM**) or iCache (**AIC**), transferred by the instruction bus (**AB**), and processed by the core (**AC**), can be manipulated by inducing bit flips into the opcode or the operands [10, 16]. Here, one possible attack would be to skip an instruction by flipping the opcode from a jump (`jalr`) to a `nop`. Similarly, by manipulating the operand, e.g., flipping `jalr 0(x5)` to `jalr 0(x6)`, the control-flow can also be arbitrarily redirected.

4.2 OpenTitan Countermeasures

Hardware-based Countermeasures. OpenTitan protects security-critical data throughout most of its life cycle using an error correction code (ECC). More specifically, the integrity of data is protected

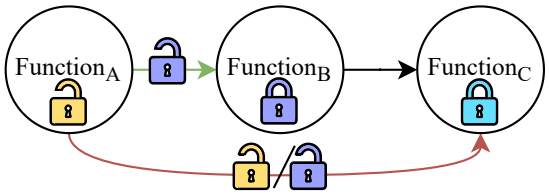


Figure 2: Encrypted call graph.

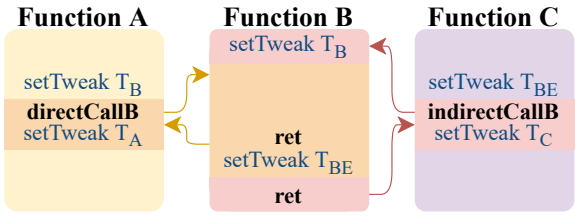


Figure 3: Instrumentation of direct and indirect calls. The different colors highlight code blocks encrypted with different tweaks.

in the data memory (**AM**), the data bus (**AB**), as well as in the register file (**AR**) of Ibex. An integrity error allows OpenTitan to detect bit-flips in **DT2** and **DT1.2a**. Additionally, ECC is used in the instruction cache (**AIC**), the program memory (**AM**), as well as the instruction bus (**AB**) to detect fault-induced bit-flips, protecting **DT3** and **DT1.1**. To prevent manipulations of the program counter (**DT1.3**), Ibex recalculates the derived program counter and triggers an error on a mismatch.

Finally, OpenTitan also provides the possibility of instantiating the Ibex core twice in a lockstep mode. Here, the execution of the second core is delayed by some cycles and the outputs of the core, i.e., the data and instruction interface outputs, are compared. Although this strategy provides strong protection against faults induced into the pipeline (**AC**), it also more than doubles the area of the CPU.

Software-based Countermeasures. In addition to the hardware-based countermeasures, the OpenTitan project also provides software-based fault protection mechanisms that are integrated into a modified LLVM toolchain. Programs compiled with this toolchain automatically store the return address into a shadow stack. In the function epilogue, before the return, the current return address is compared to the return address stored in the shadow stack, mitigating bit-flips in return addresses (**DT1.2b**). Furthermore, after each indirect branch and return, the compiler inserts an illegal instruction that triggers an exception. This strategy hinders the adversary from redirecting the control-flow by skipping these instructions (**DT1.3**, **DT3**, and **AC**).

5 SCRAMBLE-CFI

As shown in the previous section, current OpenTitan countermeasures either induce large area overheads, i.e., the dual core lockstep approach, or only provide limited protection against fault-induced control-flow manipulations when targeting the core (**AC**). To that end, in this section, we introduce SCRAMBLE-CFI, our control-flow integrity scheme that enhances the resilience of OpenTitan against these attacks with a minimal area and runtime overhead. Afterwards, in Section 6, we then highlight security guarantees and compare SCRAMBLE-CFI to existing countermeasures.

5.1 Overview

In SCRAMBLE-CFI, each function is assigned an encryption tweak during program compilation. Before execution, when loading the program into flash, the code blocks are encrypted with the corresponding tweak. At runtime, before each function call, the decryption tweak for the call target is placed into a CPU register. As the modified OpenTitan scramble engine incorporates the content of this register into the decryption, the instructions only can be decrypted when the active tweak matches the tweak determined at compile-time. Figure 2 shows the SCRAMBLE-CFI's encrypted call graph, i.e., a graph comprising all valid transactions from one function to another. When redirecting the control-flow from the current execution context to another function encrypted with a different tweak, garbled instructions are fetched and the decoding fails with a high probability. More specifically, as SCRAMBLE-CFI assigns each function, for programs without indirect branches, a unique tweak, any cross-function control-flow manipulation fails. For programs containing indirect branches, SCRAMBLE-CFI guarantees that the attacker cannot redirect the control-flow outside of the call graph. Summarized, the processor only can execute code blocks when the corresponding decryption tweak is active.

5.2 Program Instrumentation

In order to decrypt the code of a called function, the corresponding decryption tweak needs to be loaded into the tweak register before the control-flow edge, i.e., direct and indirect branches. This program instrumentation is done fully automatically in SCRAMBLE-CFI by a modified LLVM [7] RISC-V compiler.

Our custom compiler consists of an analysis and instrumentation pass operating in the backend of the toolchain. This pass first performs a program analysis to construct the call graph of the program to protect. Here, we scan each function for direct and indirect calls and determine the call target. While LLVM already provides this information for direct calls, indirect calls require a points-to analysis [8] to reveal the set of potential called functions.

After the extraction of the call graph, we assign each function a unique encryption tweak. Depending on the number of functions, this tweak is either a 5 bit or 20 bit random number.

Figure 3 shows the instrumentation of direct and indirect calls conducted by the SCRAMBLE-CFI compiler. For the direct call from function `A` to `B`, we set the tweak to the tweak of function `B`, i.e., T_B. When returning from this function, the tweak is set back to the tweak used by function `A`, i.e., T_A. Similarly, for indirect branches, e.g., from `C` to `B`, we also set and reset the tweak before and after the call. However, instead of using the same tweak as for the direct call, i.e., T_B, we add an additional entry point to the function, which is encrypted with a different tweak, i.e., T_{BE}. In this entry point, we again update the tweak to the tweak for the function to T_B. Moreover, we add a second exit point which is taken when the

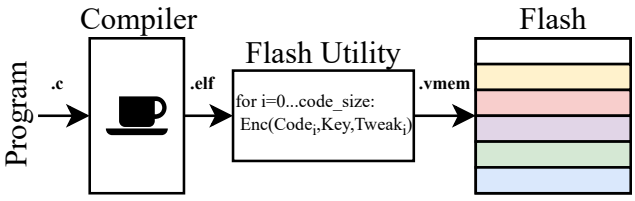


Figure 4: Deployment of protected programs. Code blocks in flash memory encrypted with different SCRAMBLE-CFI tweaks are highlighted with different colors.

function was called by an indirect branch. In this exit point, we set back the tweak to the tweak of the entry point, i.e., T_{BE}. The compiler rewrites all addresses used by direct branches to point to the instruction after the added entry point. Furthermore, in the entry point, we set a flag indicating whether the function returns with the default or the added exit point.

Adding entry points to the program is necessary because indirect calls can have multiple call targets that need to be encrypted with the same tweak. Without this additional entry point, also direct calls would need to be encrypted with this tweak, allowing the adversary to redirect a direct branch to another function which is called by an indirect branch and is outside of the call graph.

Listing 1: Tweak update instruction sequence.

```
#5 bit tweak:
 csrrwi  x0, csr_tweak, tweak_5bit
#20 bit tweak:
 lui     x28, tweak_20bit
 csrrw   x0, csr_tweak, x28
```

Listing 1 shows the instruction sequence used to set the 5 bit or 20 bit tweak into the tweak control and status register (CSR). For programs with 32 or fewer functions, a single `csrrwi` instruction loading the 5 bit `tweak` from the immediate field into the CSR `csr_tweak` is sufficient.

5.2.1 Alignment to Encryption Granular. As indicated in Figure 3, the next instruction after updating the tweak is already decrypted with this tweak. Since the decryption granularity of the underlying cipher, i.e., 64 bit for PRINCE, does not match the natural RISC-V instruction alignment of 16 or 32 bit, we need to align the set tweak instruction sequence to the decryption granularity. We conduct this alignment by padding these instructions with `nops` to 64 bit.

5.2.2 Metadata Section. The tweak for each code block is stored in a custom ELF section generated by our toolchain in the `AsmPrinter` stage. Here, the compiler emits the start address of each function and the corresponding number of basic-blocks. Moreover, the offset of each basic-block and the corresponding decryption tweak is emitted into the metadata section by the compiler. This section is then processed during program deployment.

5.3 Program Deployment

Figure 4 shows the deployment of a program protected with SCRAMBLE-CFI on OpenTitan. Our toolchain first compiles the C source code of the program to an instrumented ELF binary. Then, the flash utility program converts this binary into a `VMEM` file, which is loaded into flash memory.

As at the time of writing this paper, the OpenTitan project only provides hardware support for flash scrambling but not the necessary software support and, therefore, disables the scrambling, we extended the flash utility program to encrypt code with the PRINCE cipher. Here, we encrypt each 64 bit word with the PRINCE cipher using the flash scramble key and the SCRAMBLE-CFI tweak. Note that the flash scramble key is stored inside the one-time programmable (OTP) memory. To retrieve the SCRAMBLE-CFI tweak for each code block, the flash utility tool parses the custom metadata section emitted by our toolchain. Afterwards, the flash is initialized with the encrypted `VMEM` file and OpenTitan starts the execution of the encrypted code.

5.4 Hardware Changes

To realize SCRAMBLE-CFI on OpenTitan, minimal-intrusive hardware changes are required: First *(i)*, additional control and status registers (CSRs) need to be added to the Ibex processor. We implement these registers by using the shadow register primitives provided by the OpenTitan project. As these shadow registers duplicate the registers and compare the stored values, the content of these CSRs are protected from faults. By writing to the CSRs, software, i.e., binaries compiled with our custom toolchain, can set the current active decryption tweak as well as the lower and upper address bound. The address range registers comprise the lowest and highest address of program code, which needs to be protected by SCRAMBLE-CFI. Enabling the tweak only for a certain address range is required to access data, such as globals, also stored in the flash memory. Second *(ii)*, the tweak needs to be incorporated into the encryption primitive. One possibility would be to extend the PRINCE cipher to a tweakable block cipher with the `TWEAKEY` [4] framework. However, the required `TWEAKEY` key schedule logic would increase the complexity and area consumption of the cipher. Moreover, as SCRAMBLE-CFI does not need cryptographic strength for control-flow integrity, we inject the tweak into the key using a `XOR` operation. This exclusive or is conducted when the address sent to the flash controller is between the lower and upper address stored in the added CSRs. Otherwise, the tweak is set to 0 and the PRINCE cipher uses the default key provided by the OTP controller. Third *(iii)*, when the decryption tweak is changed by writing to the CSR, we flush the instruction cache to avoid that scrambled cached instructions are executed.

6 SECURITY ANALYSIS & COMPARISON

When the tweak register contains the tweak of the current function, the execution of code fails when the control-flow is redirected to any other function encrypted with a different encryption tweak. Before a function call, SCRAMBLE-CFI updates the tweak register with the tweak of the called function. Then, the control-flow can only be redirected to this function or functions encrypted with the same tweak. In SCRAMBLE-CFI, the entry points (cf. Section 5.2) of functions that an indirect branch can call share the same encryption tweak. This is inevitable as, at compile-time, the toolchain cannot determine which function gets called by an indirect call at runtime. However, indirect calls are rare in typical programs and

the attacker only can redirect the control-flow to functions that can be reached with this indirect call, i.e., are within the call graph. For programs without indirect calls, SCRAMBLE-CFI assigns all functions a unique encryption tweak. Summarized, for any control-flow redirection outside of the call graph (**CFM1**) the wrong decryption tweak is deterministically used for programs that contain less than 2^{20} functions. For programs that contain more functions than the available tweak space, tweak collisions can occur.

When fetching instructions from program memory with a wrong decryption tweak, garbled instructions are retrieved. With a high probability, the decoding of these instructions in the instruction decoder pipeline stage fail and an exception is triggered. However, it could be possible that a garbled instruction again forms a valid instruction. Nevertheless, *(i)* the probability that the subsequent instructions are also valid is low and *(ii)* usually, the attacker aims to execute a certain instruction in the function and not any that does not trigger an exception.

Note that SCRAMBLE-CFI cannot prevent an adversary from manipulating conditional branches (**CFM2**) or from redirecting the control-flow within a function from one basic-block to another (**CFM3**). However, SCRAMBLE-CFI could be extended to provide fine-granular protection for highly security-critical code blocks by encrypting code blocks within a function with different encryption keys.

In the current prototype of SCRAMBLE-CFI, we incorporate the tweak into the encryption by XORing it to the encryption key. This XOR creates a dependency of key and tweak, allowing an attacker to potentially learn about the key when the tweak is known. However, the binary (including the tweaks) is inaccessible to an adversary as it is stored in the encrypted flash.

6.1 Security Comparison

Table 1 highlights protection guarantees of different hardware- and software-based OpenTitan countermeasures and SCRAMBLE-CFI against faults into different attack targets.

The ECC-based countermeasures provide full protection when inducing faults into their protection domain. For example, bit-flips induced into instructions (**DT3**) stored in the instruction cache (**AIC**) or the program memory (**AM**) can be detected reliably. As the program counter (PC) protection recalculates and compares the PC to the current PC, also faults into the core (**AC**) can be detected. The software-based defense integrated into the custom LLVM toolchain only provides partial protection, i.e., only control-flow manipulations aiming to manipulate return addresses (**DT1.2b**) or skipping certain instructions (**DT3**) can be detected. As shown in Table 1, from the existing countermeasures, only the lockstep approach can provide strong protection against faults induced into the core (**AC**). However, this strategy also induces a high area overhead as the Ibex core needs to be instantiated twice and an error detection logic needs to be added. In comparison, SCRAMBLE-CFI provides strong protection against control-flow manipulations for all attack targets, even when targeting the core, with minimal hardware overhead. Especially when combined with the other existing countermeasures, SCRAMBLE-CFI significantly minimizes the attack surface and allows OpenTitan to withstand most of the control-flow attacks. Combined, OpenTitan is protected against arbitrary control-flow manipulations (**CFM1**, **CFM2**, and **CFM3**) when targeting all attack targets except the core (**AC**). When injecting faults into **AC**, SCRAMBLE-CFI fills the protection gap of hindering an adversary from redirecting the control-flow outside of the call graph (**CFM1**). However, when also **CFM2** and **CFM3** need to be mitigated, SCRAMBLE-CFI needs to be deployed on a finer granularity (cf. Section 6) or the lockstep approach needs to be installed. Summarized, we argue that SCRAMBLE-CFI provides a good area-security tradeoff allowing OpenTitan to withstand a multitude of different fault attacks.

Table 1: Protection guarantees of different countermeasure when targeting different attack surfaces.

	Attack Targets				
Countermeasure	**AR**	**AIC**	**AB**	**AM**	**AC**
Data memory ECC	-	-	-	✔	-
Program memory ECC	-	-	-	✔	-
Data bus ECC	-	-	✔	-	-
Instruction bus ECC	-	-	✔	-	-
Register file ECC	✔	-	-	-	-
iCache ECC	-	✔	-	-	-
PC protection	✔	-	-	-	✔
SW-based defense	◐	-	-	-	◐
Dual core lockstep	✔	✔	-	-	✔
SCRAMBLE-CFI	◐	◐	◐	◐	◐

✔ Full ◐ Partial - No Protection

7 PERFORMANCE OVERHEAD

To evaluate the performance and code size overhead, we compiled the Embench-IoT [13] benchmarks with our custom toolchain. We excluded benchmarks requiring libraries, such as `math` and `string`, currently not provided by the OpenTitan framework.

To measure the code size overhead, we compared the protected binaries to the unprotected baseline with the GNU `size` utility. Our analysis shows a code size overhead between 0.74 % and 8.88 % and a geometric mean of 1.69 %. This overhead comprises the *(i)* inserted tweak switch instructions, the *(ii)* alignment of these instructions to the encryption granular, and the *(iii)* metadata binary section. As this section only is needed by the flash utility program to encrypt code blocks with different encryption tweaks (cf. Section 5.3), this metadata is not stored in the flash memory.

In order to analyze the performance impact of binaries instrumented with SCRAMBLE-CFI, we measured the CPU cycles by reading the `mcycle` CSR of Ibex. Here, we executed the protected and unprotected binaries on a cycle-accurate Verilator model of OpenTitan. As shown in Figure 5, we measured a runtime overhead between 0.22 % and 143.35 % and a geometric mean of 7.02 %. Benchmarks, such as `crc32` or `tarfind`, frequently calling small functions induce larger runtime overheads than benchmarks only performing a small number of function calls.

8 AREA OVERHEAD

The public OpenTitan hardware design flow currently only allows to synthesize the Ibex core with open-source synthesis tools. Therefore, we synthesized the Ibex processor with the Yosys open synthesis suite and the Nangate 45 nm standard cell library to analyze

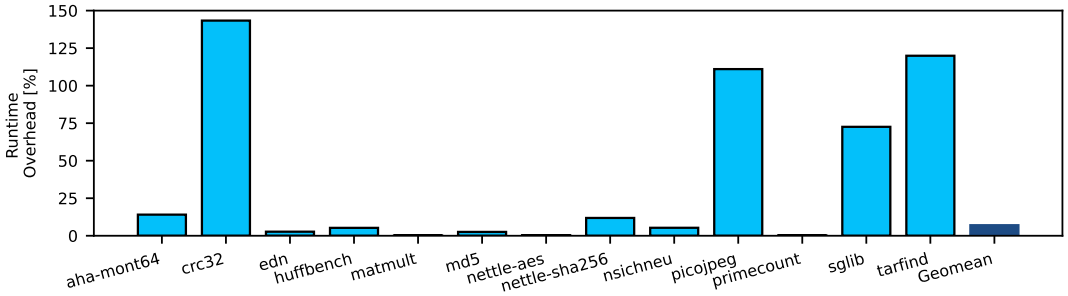


Figure 5: Runtime overhead for the Embench-IoT benchmarks.

the area overhead introduced by our hardware changes. Here, we measured an area increase from 26.48 kGE to 27.53 kGE (3.97 %). These hardware changes comprise the additional CSRs as well as the address range comparison. Note that the XOR of the tweak with the key is conducted in the flash controller and is currently not reflected in the hardware overhead number. According to the synthesis logs created with the proprietary design flow published by the OpenTitan project [12], the Ibex occupies 3.4 % of the overall chip area. Hence, the hardware overhead induced by SCRAMBLE-CFI to the overall area is negligible.

9 RELATED WORK

Currently, encryption-based control-flow integrity schemes, such as SOFIA [3] or SCFP [19], require intrusive hardware changes in the processor's pipeline to realize their protection. Hence, when integrating these changes into a chip, an extensive re-verification of the entire processor is needed. Although EC-CFI [9] provides cryptographic CFI on Intel hardware without hardware modifications, the measured runtime overheads are high. In comparison, SCRAMBLE-CFI induces small runtime overheads with minimal hardware changes that allow designers to re-verify their design with minimal effort.

10 CONCLUSION

In this paper, we thoroughly analyzed fault threat vectors allowing an adversary to manipulate the control-flow of software executed on OpenTitan. We provided an overview of current OpenTitan countermeasures and discussed their protection capabilities. Furthermore, we introduced SCRAMBLE-CFI, which mitigates fault attacks aiming to redirect the control-flow outside of the call graph. We showcased that SCRAMBLE-CFI is a strong security addition to existing countermeasures inducing minimal runtime and area overheads.

11 ACKNOWLEDGMENTS

This project has received funding from the Austrian Research Promotion Agency (FFG) via the AWARE project (grant number 891092).

REFERENCES

[1] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. 2006. The Sorcerer's Apprentice Guide to Fault Attacks. *Proc. IEEE* 94 (2006), 370–382.
[2] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. 2012. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In *ASIACRYPT (LNCS, Vol. 7658)*. 208–225.
[3] Ruan de Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. 2017. SOFIA: Software and control flow integrity architecture. *Comput. Secur.* 68 (2017), 16–35.
[4] Jérémy Jean, Ivica Nikolic, and Thomas Peyrin. 2014. Tweaks and Keys for Block Ciphers: The TWEAKEY Framework. In *ASIACRYPT (LNCS, Vol. 8874)*. 274–288.
[5] Scott Johnson, Dominic Rizzo, Parthasarathy Ranganathan, Jon McCune, and Richard Ho. 2018. Titan: enabling a transparent silicon root of trust for cloud. In *Hot Chips: A Symposium on High Performance Chips*, Vol. 194.
[6] Dusko Karaklajic, Jörn-Marc Schmidt, and Ingrid Verbauwhede. 2013. Hardware Designer's Guide to Fault Attacks. *IEEE Trans. Very Large Scale Integr. Syst.* 21 (2013), 2295–2306.
[7] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. 75–88.
[8] Pascal Nasahl, Robert Schilling, and Stefan Mangard. 2021. Protecting Indirect Branches Against Fault Attacks Using ARM Pointer Authentication. In *HOST*. 68–79.
[9] Pascal Nasahl, Salmin Sultana, Hans Liljestrand, Karanvir Grewal, Michael LeMay, David M. Durham, David Schrammel, and Stefan Mangard. 2023. EC-CFI: Control-Flow Integrity via Code Encryption Counteracting Fault Attacks. *CoRR* abs/2301.13760 (2023).
[10] Pascal Nasahl and Niek Timmers. 2019. Attacking AUTOSAR using software and hardware attacks. *escar USA* (2019).
[11] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Trans. Reliab.* 51 (2002), 111–122.
[12] OpenTitan. 2023. CHIP_EARLGREY_ASIC Synthesis Results. https://reports.opentitan.org/hw/top_earlgrey/syn/2022.07.02_00.42.20/results.html.
[13] David Patterson, Jeremy Bennett, Palmer Dabbelt, Cesare Garlati, G. S. Madhusudan, and Trevor Mudge. 2023. Embench: Open Benchmarks for Embedded Platforms. https://www.embench.org/.
[14] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. 2005. SWIFT: Software Implemented Fault Tolerance. In *CGO*. 243–254.
[15] Robert Schilling, Pascal Nasahl, and Stefan Mangard. 2022. FIPAC: Thwarting Fault- and Software-Induced Control-Flow Attacks with ARM Pointer Authentication. In *COSADE (LNCS, Vol. 13211)*. 100–124.
[16] Niek Timmers, Albert Spruyt, and Marc Witteman. 2016. Controlling PC on ARM Using Fault Injection. In *FDTC*. 25–35.
[17] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sébastien Ermeneux. 2020. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot-Extended Version. *IEEE Trans. Computers* 69 (2020), 1449–1459.
[18] Ingrid Verbauwhede, Dusko Karaklajic, and Jörn-Marc Schmidt. 2011. The Fault Attack Jungle - A Classification Model to Guide You. In *FDTC*. 3–8.
[19] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. 2018. Sponge-Based Control-Flow Protection for IoT Devices. In *EURO S&P*. 214–226.