





# Control Flow Protection by Cryptographic Instruction Chaining

Shahzad Ahmad<sup>1</sup><sup>a</sup>, Stefan Rass<sup>1,2</sup><sup>b</sup>, Maksim Goman<sup>1</sup><sup>c</sup>, Manfred Schlägl<sup>1,3</sup><sup>d</sup> and Daniel Große<sup>3</sup><sup>e</sup>

<sup>1</sup>*LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Altenbergerstrasse 69, 4040 Linz, Austria*

<sup>2</sup>*Institute for Artificial Intelligence and Cybersecurity, Alpen-Adria-University Klagenfurt, Universitätsstrasse 65-67, 9020 Klagenfurt, Austria*

<sup>3</sup>*Institute for Complex Systems, Johannes Kepler University Linz, Altenbergerstrasse 69, 4040 Linz, Austria*  
{shahzad.ahmad, stefan.rass, maksim.goman, manfred.schlaegl, daniel.grosse}@jku.at

**Keywords:** Secure execution environment, Instruction confidentiality, Control flow integrity, Key derivation chains, Authenticated encryption, Execution order verification, Chosen-instruction attacks

**Abstract:** We present a novel secure execution environment that provides comprehensive protection for program execution through a unified cryptographic approach. Our construction employs authenticated encryption, ensuring instruction confidentiality, integrity, and correct execution ordering. The system’s key innovation lies in its cryptographic binding of consecutive instructions through a novel key chaining mechanism that prevents instruction reordering and replay attacks while maintaining an enforced order of instructions using cryptographic chaining via keys. We introduce specialized handling for control flow operations, including branches, jumps, and function calls, that preserves security guarantees across complex execution paths. The framework includes a protection mechanism for registers and memory, creating a fully secured execution environment. Our performance analysis quantifies the computational overhead and provides a Python proof-of-concept implementation that validates the practical viability of our approach <https://github.com/shahzadssg/Control-Flow-Protection-by-Cryptographic-Instruction-Chaining.git>.

## 1 INTRODUCTION

The security of program execution remains a critical challenge in modern computing systems as threats grow increasingly sophisticated. Traditional protection mechanisms, such as memory isolation and access control, are inadequate against advanced attacks exploiting speculative execution, instruction reordering, and memory tampering [Abadi et al., 2009, Kocher et al., 2018]. This paper presents a novel secure execution environment that addresses these vulnerabilities through a unified cryptographic approach, ensuring instruction confidentiality, integrity, and correct execution ordering.


Secure program execution requires addressing three interrelated challenges: maintaining instruction confidentiality to prevent leakage of sensitive behavior, ensuring execution integrity to protect against tampering, and enforcing strict control over instruc-


tion order to thwart reordering attacks. These issues are critical in systems processing sensitive algorithms, where vulnerabilities like return-oriented programming (ROP) [Shacham, 2007] and speculative execution attacks (e.g., Spectre [Kocher et al., 2020]) exploit execution patterns and microarchitectural side channels to compromise security.


Previous approaches, such as memory encryption [Gueron, 2016], control-flow integrity (CFI) [Wang et al., 2012], and hardware-based solutions like Intel SGX [Costan and Devadas, 2016] and Morpheus [Gallagher et al., 2019], offer partial protections but lack a unified framework. Memory encryption secures data but not execution order, CFI validates paths without ensuring confidentiality, and hardware solutions rely on specific features without inherent ordering guarantees. These gaps leave systems vulnerable to multi-vector attacks, necessitating a comprehensive solution that integrates all required security properties with practical performance.


Our contribution is a secure execution environment that combines instruction confidentiality, integrity protection, and execution order verification through a cryptographic framework. This is achieved

<sup>a</sup>  <https://orcid.org/0000-0002-9654-869X>

<sup>b</sup>  <https://orcid.org/0000-0003-2821-2489>

<sup>c</sup>  <https://orcid.org/0000-0002-7735-7409>

<sup>d</sup>  <https://orcid.org/0009-0007-3275-4797>

<sup>e</sup>  <https://orcid.org/0000-0002-1490-6175>

via a novel key chaining mechanism that binds consecutive instructions, preventing reordering and replay attacks, and specialized handling of control flow operations (e.g., branches, jumps, function calls) to maintain security across complex paths. A Python proof-of-concept validates its feasibility, supported by performance and security analyses.

### 1.1 Threat Model and Assumptions

Our threat model assumes adversaries can observe and manipulate encrypted instructions and memory, execute chosen-instruction attacks, and run arbitrary code outside the secure environment, but cannot exploit microarchitectural side channels due to serialized processing. We assume a secure processor with a protected master key, symmetric encryption, HMAC, and secure key generation capabilities, enabling guarantees against confidentiality breaches, integrity violations, and control-flow hijacking.

## 2 RELATED WORK

Secure execution has become critical in protecting modern computing systems against increasingly complex attack vectors. Existing approaches include memory encryption, control-flow integrity, processor architecture enhancements, a cryptographic computation framework, and hardware-based security solutions. This section reviews key developments in these areas, highlighting the limitations that our work aims to address through a cryptographically unified secure execution environment.

### 2.1 Memory Encryption and Smart Card Security

Memory encryption has been widely studied to protect data confidentiality and integrity, particularly for environments vulnerable to physical access or side-channel attacks. Henson and Taylor [Henson and Taylor, 2014] provided an extensive survey on memory encryption techniques, describing strategies such as block ciphers and integrity verification. Specialized implementations, such as the low-latency memory encryption for smart cards presented by Ege et al. [Ege et al., 2011], showcase memory encryption tailored for constrained devices, balancing security with efficiency.

### 2.2 Control-Flow Integrity (CFI) and Instruction Set Randomization (ISR)

Control-flow integrity (CFI) and Instruction Set Randomization (ISR) have proven effective in defending against code-reuse attacks and code injection attacks. The foundational work of Abadi et al. [Abadi et al., 2009] on CFI introduced a robust approach for securing program control flow against hijacking attempts. Tice et al. [Tice et al., 2014] expanded CFI implementation to widely used compilers like GCC and LLVM, creating practical tools for protecting against forward-edge attacks. ISR techniques, such as those proposed by Kc et al. [Kc et al., 2003], further mitigate code injection by randomizing instruction encoding at runtime, preventing predictable exploit payloads. Meanwhile, Christou et al. [Christou et al., 2020] explored architectural enhancements to support ISR, ensuring that randomized instruction sets are resilient to static analysis and reverse engineering. Werner et al. [Werner et al., 2018] propose sponge-based control-flow integrity for IoT, leveraging lightweight cryptography. Savry et al. [Chamelot et al., 2022] integrate instruction and data authenticated encryption (Confidaent) for holistic protection.

### 2.3 Data-Oblivious and Encrypted Computation

Data-oblivious computation frameworks have emerged to facilitate secure computations on untrusted platforms by concealing data access patterns. PHANTOM, proposed by Maas et al. [Maas et al., 2013], utilizes oblivious RAM to secure memory accesses, ensuring that patterns reveal minimal information to potential adversaries. Similarly, Liu et al. [Liu et al., 2015] developed OblivVM, a programming framework for oblivious computation, offering support for cryptographic protocols and secure multiparty computation. Rass et al. [Rass et al., 2015] introduced Oblivious Lookup-Tables (OLUTs), enabling arbitrary function evaluation on encrypted data using group homomorphic encryption through clever use of Vandermonde matrices, offering an efficient middle ground between basic homomorphic schemes and fully homomorphic encryption (FHE). HEROIC, introduced by Tsoutsos and Maniatakos [Tsoutsos and Maniatakos, 2014], implemented homomorphic encryption within a minimalistic one-instruction architecture, proving useful for encrypted cloud computing and privacy-preserving processing. However, these frameworks

often need more formulation which come at a higher computational cost.

## 2.4 Side-Channel and Cache Attack Mitigation

As side channel attacks become more sophisticated, research has turned to mitigating vulnerabilities in cache and timing. Gruss et al. [Gruss et al., 2015] presented cache template attacks, demonstrating how adversaries can exploit last-level cache inclusivity for information leakage. Speculative execution vulnerabilities, as exposed by Kocher et al. [Kocher et al., 2020] with the Spectre attacks, further underscored the need for more rigorous protections at the hardware level. In response, Rass and Schartner [Rass and Schartner, 2016] examined countermeasures against chosen instruction attacks within cryptographic environments, proposing instruction verification to prevent unauthorized instruction modification.

## 2.5 Cryptographic Techniques for Execution Order and Integrity

Recent research has focused on cryptographic methods to enforce correct instruction sequencing and program integrity. Zahur et al. [Zahur et al., 2015] introduced a half-gate approach to reduce data transfer in garbled circuits, improving efficiency in cryptographic computation. Zahur and Evans [Zahur and Evans, 2015] developed Obliv-C, a language for extensible data-oblivious computation, enabling flexible applications in privacy-preserving environments. The formal verification of control-flow graph flattening by Blazy and Trieu [Blazy and Trieu, 2016] highlights ongoing efforts to ensure control-flow integrity within cryptographic environments, preventing adversarial manipulation of execution order. De Clercq et al. [de Clercq et al., 2017] (SOFIA) combine software and control-flow integrity for embedded systems. Stecklina et al. [Stecklina et al., 2015] introduced Intrinsic Code Attestation, which uses cryptographic instruction chaining for embedded devices to prevent unauthorized code execution and instruction reordering attacks. Their work demonstrated the feasibility of creating cryptographic dependencies between consecutive instructions using block ciphers, providing early validation of instruction chaining concepts in resource-constrained environments.

Our approach leverages key aspects from prior works, introducing a cryptographically unified secure execution environment that combines data confidentiality, integrity, and execution order verification. This work extends current models by addressing forward

and backward secrecy in instruction sequences, offering tamper detection and replay protection against sophisticated attacks. The unique key mechanism in our approach not only enhances security but also simplifies the execution process. Like Morpheus II [Harris et al., 2021], we employ strong cryptography to protect code and control data. However, we advance beyond simple encryption by creating a cryptographic binding between consecutive instructions through our key chaining mechanism.

# 3 CONSTRUCTION

We present a novel secure execution environment that provides confidentiality and integrity protection for instructions and data while ensuring correct execution order even in the presence of adversarial tampering attempts. The system employs a combination of symmetric encryption, HMAC, and a unique key chaining mechanism to create a provably secure execution model.

## 3.1 System Model and Notation

We consider a processor architecture with the following essential components. The system uses a set of registers  $R = \{r_1, r_2, \dots, r_n\}$  alongside an instruction memory  $I$  and data memory  $M$ , both storing encrypted content with authentication tags. Program execution is tracked via an instruction pointer  $ip \in \mathbb{N}$ , while an encrypted call stack manages function calls and returns. The architecture’s security relies on cryptographic primitives including symmetric encryption  $E$ , decryption  $D$ , an HMAC function  $H$ , and a secure random key generator  $KeyGen$ . We use the following notation:

- $K_M$ : The Master key is known only to the secure processor. The complete life cycle of this key (including creation, distribution, usage, rotation and destruction processes) is outside the scope of this paper. We assume secure processes are in place to manage this key.
- $K_{prev}$ : Key used to decrypt registers before executing an instruction
- $K_{next}$ : Key used to encrypt registers after executing an instruction
- $K_{ret}$ : Randomly generated key to be put onto the stack
- $E_K(m)$ : Symmetric encryption of message  $m$  under a key  $K$

- $D_K(c)$ : Symmetric decryption of ciphertext  $c$  under a key  $K$
- $H_K(m)$ : HMAC of message  $m$  under key  $K$
- $KeyGen(t) : \{0,1\}^t$ , a cryptographically secure random key generator that produces a key of length  $t$ , where  $t$  is the security parameter

### 3.2 Cryptographic Chaining and Instruction Execution

Our system employs a key chaining mechanism to ensure the correct order of instruction execution and to prevent various attacks, including instruction reordering and replay attacks. This mechanism creates a cryptographic binding between consecutive instructions, effectively forming a chain of keys that must be followed for correct execution. The key derivation chain operates as follows:

#### 3.2.1 Initialization

For the first instruction  $I_0$

- $K_{prev} \leftarrow K_M$  (the Master key)
- $K_{next} \leftarrow KeyGen(t)$  (a randomly generated key, independently generated afresh for each instruction)

#### 3.2.2 Sequential Instructions

These basic arithmetic, logic, and data movement instructions within the register file (e.g. MOV) execute in sequence without altering the control flow. These instructions maintain the strict sequential execution order through the cryptographic chaining mechanism, where the previous key of each instruction must correspond to its predecessor's next key. For each sequential instruction  $I_i$  (where  $i > 0$ ):

The key mechanism operates as follows:

- $K_{prev} \leftarrow K_{next}$  of instruction  $I_{i-1}$  (the previous instruction's next key)
- $K_{next} \leftarrow KeyGen(t)$  (a freshly generated random key)

The key derivation chain works in conjunction with the instruction encoding scheme. Each instruction is encoded as a tuple:

$$I_i = (op_i, args_i, K_{prev,i}, K_{next,i})$$

Each instruction  $i$  comprises four essential elements: the opcode ( $op_i$  is encoded as 8 bits) that defines the operation, instruction arguments ( $args_i$ ) as 32-bit values (though other bit lengths are supported),  $K_{prev,i}$  (128 bits) for decrypting registers before execution, and  $K_{next,i}$  (128 bits) for encrypting registers

after execution. This structure ensures secure instruction handling throughout the execution cycle. The instruction is then processed and stored as:

$$T_i = H_{K_M}(I_i)$$

$$C_i = E_{K_M}(I_i || T_i)$$

where  $T_i$  is the authentication tag for instruction  $I_i$ , and  $||$  denotes concatenation.

#### Execution Process

The execution of a sequence of instructions proceeds as follows:

- i) Fetch encrypted instruction  $C_i$  at address  $ip$
- ii) Decrypt and verify integrity:
  - $(I'_i || T'_i) = D_{K_M}(C_i)$
  - $T_i^* = H_{K_M}(I'_i)$
  - If  $T_i^* \neq T'_i$ , halt execution
- iii) Parse  $I'_i = (op_i, args_i, K_{prev}, K_{next})$
- iv) Decrypt registers: For each register  $j \in \{1, \dots, n\}$ ,  $r_j = D_{K_{prev}}(R[j])$ . In this register system,  $j$  serves as the register index within the total set of  $n$  registers in the bank. The system maintains two representations for each register:  $R[j]$  represents its encrypted state, while  $r_j$  holds the corresponding decrypted value used during computation. This dual representation enables both secure storage and practical usage of register contents.
- v) Execute instruction  $(op_i, args_i)$  on decrypted registers
- vi) Update state: registers and memory
- vii) Encrypt updated registers: For each register  $j$  in  $\{1, \dots, n\}$ :  $R[j] = E_{K_{next}}(r_j)$
- viii) Update  $ip$  (instruction pointer) based on instruction type.

In our model, cryptographic chaining enforces a strict sequential order across all pipeline stages: fetch, decode, execute, and commit. Each instruction's decryption and validation depends on the cryptographic key from its predecessor, serializing the entire pipeline. Unlike traditional out-of-order architectures where only commit/retirement stages enforce program order, our scheme restricts parallelism at every stage. Functional units cannot process instructions out-of-order, as the cryptographic binding requires the prior instruction's key to unlock subsequent operations. This ensures that even if dependencies theoretically allow reordering, the cryptographic chain mandates sequential progression.

### 3.2.3 Conditional Branches

Our secure execution environment handles conditional branches through a deterministic key derivation mechanism that maintains cryptographic continuity across both potential execution paths.

For a conditional branch instruction  $I_{br}$

- $K_{prev} \leftarrow K_{next}$  of the previous instruction
- $K_{next\_false} \leftarrow KeyGen(t)$  if the branch is “non-taken” (sequential)
- However, if the branch is “taken” then for the branch instruction’s  $K_{next\_true}$ , the function must have properties:
  - i) the output must be suitable for use as an encryption key, typically a fixed-length bit string matching the security parameter  $t$
  - ii) the computation must be deterministic but depend on both the target label and the master key  $K_M$  to prevent adversaries from computing it themselves. This can be expressed as:  $K_{next\_true} \leftarrow func(target\_label) = H_{K_M}(target\_label)$

A *target\_label* represents a secure destination identifier within the execution environment that serves multiple critical purposes in the control flow mechanism. It is an integral component encoded within various control transfer instructions (conditional branches, unconditional jumps, and function calls) that specifies where program execution should continue. The *target\_label* is both encrypted and integrity-protected through HMAC verification, and plays a crucial role in the cryptographic key derivation process through the function  $func(target\_label)$ . This design ensures that only authorized code possessing the master key can compute valid keys for execution at target locations. The *target\_label* maintains the cryptographic binding between source and destination instructions while enabling secure control transfers, as it serves as the basis for deriving the next encryption key ( $K_{next}$ ) that will be used to verify proper execution sequencing at the destination. This mechanism prevents unauthorized jumps or execution reordering while allowing legitimate program control flow to proceed securely within the encrypted execution environment.

The above key derivation for  $K_{next\_true}$  provides critical security properties. First and foremost, only entities possessing the master key  $K_M$  can compute  $K_{next\_true}$ , ensuring that key derivation remains restricted to authorized parties within the secure execution environment. The construction’s deterministic nature means that identical labels consistently produce the same keys, which is essential for maintaining

execution consistency across multiple runs of the program. Additionally, the output maintains the cryptographic operations. The use of HMAC ensures that different labels produce cryptographically independent keys, preventing any potential key-relation attacks. This independence property is crucial for maintaining security isolation between different branches in the program’s execution flow.

The conditional branch instruction is encoded as:

$$I_{br} = (BR_{cond}, target\_label, K_{prev}, K_{next\_true}, K_{next\_false})$$

The conditional branch instruction consists of several key components that work together to ensure secure execution flow. The instruction uses  $BR_{cond}$  to represent a generic conditional branch operation that can handle any type of branching logic. The *target\_label* specifies the destination address in the program’s code where execution should proceed after the control flow transfer, whether it’s from a jump, branch, or function call. To maintain security during branching, the system employs two distinct keys:  $K_{next\_true}$  for when the branch is taken and execution moves to the target address, and  $K_{next\_false}$  for when the branch is not taken and execution continues sequentially. This dual-key structure ensures cryptographic separation between the two possible execution paths.

### Execution Process

The execution of a conditional branch instruction proceeds as follows:

$$K_{next} \leftarrow \begin{cases} K_{next\_true} & \text{if branch taken} = True \\ K_{next\_false} & \text{if branch not taken} = False \end{cases}$$

$$ip = \begin{cases} target\_label & \text{if branch taken} = True \\ ip + 1 & \text{if branch not taken} = False \end{cases}$$

When execution continues at a *target\_label* location, the key chaining mechanism ensures proper sequential execution 3.2.2 resumes from that point.

The instruction format for sequential instructions is

$op, args, K_{prev}, K_{next} = 8 + 32 + 128 + 128 = 296$  bits and for conditional branch it includes a *target\_label* (32 bits) and

$K_{next\_true}, K_{next\_false}$  ( $2 \times 128$  bits). In order to mitigate the memory overhead, the deterministic keys (e.g.,  $H_{K_M}(target\_label)$ ) reduces storage for branch targets and in future we will explore in compressing authentication tags or using shorter keys (e.g., 64-bits truncated HMACs).

### 3.2.4 Unconditional Jumps (JMP)

Unlike conditional branches, unconditional jumps always transfer control to their target location. The key management must be such that the key for the next instruction is computable from any point in the program, i.e., we let this key depend on the jump target and the secret master key to avoid an attacker computing it from the source code already.

For an unconditional jump instruction  $I_{jmp}$

- $K_{prev} \leftarrow K_{next}$  of the previous instruction
- $K_{next} \leftarrow func(target\_label) = H_{K_M}(target\_label)$

The jump instruction is encoded as:

$$I_{jmp} = (JMP, target\_label, K_{prev}, K_{next})$$

The unconditional jump instruction consists of three core components that enable secure control flow transfer. The *JMP* opcode designates an unconditional jump operation that transfers execution to a new location, while the *target\_label* specifies the exact destination address where execution will continue. The third component,  $K_{next}$ , is a cryptographic key derived deterministically from the target label to maintain security throughout the jump operation.

The keychain operation for jumps follows a carefully designed sequence to ensure security. First,  $K_{next}$  is computed deterministically using the  $func(target\_label)$  function. This deterministic derivation ensures that any jump targeting the same label will generate identical key values, providing consistency in the security mechanism. Finally, this approach creates a robust cryptographic binding between jump instructions and their target locations, preventing unauthorized modifications to the control flow. This three-part mechanism works together to maintain both the integrity and security of program execution during unconditional jumps.

#### Execution Process

The execution of an unconditional jump instruction proceeds as follows:

- i) Encrypt registers with  $K_{next} \leftarrow func(target\_label)$
- ii) Transfer control to *target\_label* location
- iii) Target instruction must use jump's  $K_{next}$  at its  $K_{prev}$

### 3.2.5 Function Calls

Our secure execution environment implements function calls using a deterministic key derivation mechanism for function entry points, ensuring consistent access across multiple calls. Furthermore, each CALL

instruction is assumed to have a corresponding RET (return) instruction.

For a function call instruction (*CALL*):

- $K_{prev} \leftarrow K_{next}$  of the previous instruction
- $K_{next} \leftarrow func(target\_label)$  is deterministically derived from the target label
- $K_{ret} \leftarrow KeyGen(t)$

The *CALL* instruction is encoded as a tuple:

$$I_{call} = (CALL, target\_label, K_{prev}, K_{next})$$

#### Execution Process

- i) Registers are decrypted using the  $K_{prev}$  of the *CALL* instruction
- ii) The return address ( $ip_{ret}$ ) is set to the address of the next instruction after the call function is performed
- iii) The  $K_{next}$  of the *RET* instruction is denoted by  $K_{ret}$ , and it is put on the stack (together with  $ip_{ret}$ ) and will be used as  $K_{prev}$  for the instruction where execution returns.
- iv) The return information is encrypted and pushed onto the call stack:
$$S_{push} = E_{K_M}(ip_{ret} \parallel K_{ret} \parallel H_{K_M}(ip_{ret} \parallel K_{ret}))$$
- v) Registers are encrypted using the  $K_{next}$  of the *CALL* instruction, and this next key will become the previous key for the called function
- vi) The instruction pointer is set to the target address of the called function.

This approach ensures that the return address and key are securely stored and cannot be tampered with during the function execution.

### 3.2.6 Function Returns

The return mechanism (*RET* instruction) is designed to securely restore the execution context and maintain the integrity of the key chain.

For a return instruction:

- $K_{prev} \leftarrow K_{next}$  of the previous instruction
- $K_{next}$  is obtained from the call stack

The *RET* instruction is encoded as:

$$I_{ret} = (RET, K_{prev}, *)$$

where \* indicates  $K_{next}$  will be determined at runtime from the call stack.

## Execution Process

- i) Call Stack Retrieval: The encrypted return information is popped from the call stack:

$S_{pop}$  = top element of the call stack

- ii) The return information is decrypted and verified:

$$(ip_{ret}, K_{ret}, T) = D_{K_M}(S_{pop})$$

$$T^* = H_{K_M}(ip_{ret} || K_{ret})$$

If  $T^* \neq T$ , execution is halted due to potential tampering.

- iii) The instruction pointer is set to  $ip_{ret}$ .
- iv)  $K_{next}$  is set to  $K_{ret}$ .
- v) Registers are encrypted using  $K_{ret}$  (which becomes the  $K_{prev}$  for the next instruction).

This mechanism ensures that:

- a) The return address cannot be manipulated during function execution.
- b) The key chain integrity is maintained across function calls and returns.
- c) Any tampering with the call stack or attempt to return to an unauthorized location will be detected.

By implementing function calls and returns this way, our secure execution environment maintains a continuous chain of keys throughout the program's execution, including across function boundaries. This approach provides strong protection against various attacks, including return-oriented programming (ROP) and other control-flow hijacking attempts. The key derivation chain mechanism makes sure that the correct execution order is maintained even in the presence of complex control flow structures. The use of distinct keys for different execution paths ( $K_{next\_true}$  and  $K_{next\_false}$  for branches,  $K_{ret}$  for function returns) provides strong cryptographic binding between instructions and their intended execution sequence. The key derivation chain provides several security properties:

1. *Instruction Ordering*: The chain assures that instructions can only be executed in the intended order, even across branches and function calls.
2. *Replay Protection*: Each instruction uses unique keys, preventing replay attacks.
3. *Forward Secrecy*: Compromising one instruction's keys does not reveal information about subsequent instructions.
4. *Backward Secrecy*: Similarly, compromising one instruction's keys does not allow the derivation of keys for previous instructions.

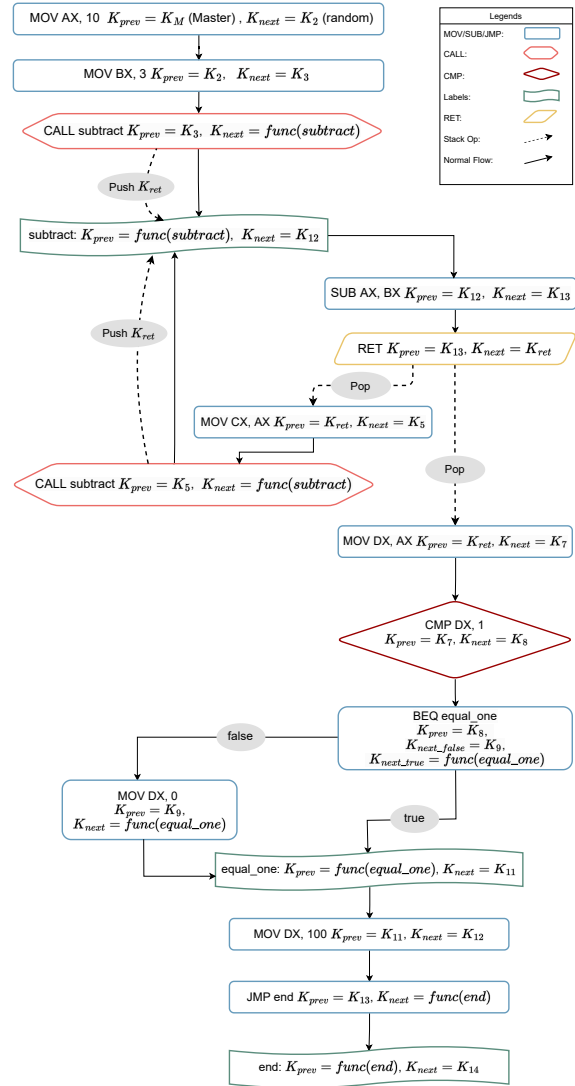


Figure 1: Execution flow example

Figure 1 presents an illustration of the execution flow in the secure execution environment, demonstrating how the key chaining mechanism works across various instruction types and control flow operations. The execution flow begins with basic instructions, starting with MOV AX, 10 that uses the master key ( $K_M$ ) as  $K_{prev}$  and generates a random  $K_2$  as  $K_{next}$ , followed by MOV BX, 3 which continues the key chain. The diagram then shows how function calls are handled, with a CALL subtract instruction that derives its  $K_{next}$  using a special function of the target label. The function call mechanism is illustrated with stack operations, showing how return keys ( $K_{ret}$ ) are pushed onto the stack, and how the subtract function executes its operations (including SUB AX, BX) before returning via a RET instruction.

The figure also demonstrates the handling of com-

plex control flow operations, including conditional branching through a CMP DX, 1 instruction followed by a BEQ equal\_one branch. This shows how the system manages different execution paths, with separate key chains maintained for both the taken and not-taken paths of the branch. The diagram includes an unconditional jump (JMP end) as well, illustrating how control flow is maintained across different types of transfers. Throughout the execution flow, the diagram uses a clear visual notation system with distinct shapes for different instruction types, directional arrows showing execution flow, and dotted lines indicating the stack operation. The key transitions between instructions are explicitly labeled, showing how  $K_{prev}$  and  $K_{next}$  values are maintained and derived throughout the program's execution. This visualization effectively demonstrates how the system maintains cryptographic continuity while handling complex control flow structures, ensuring secure execution across all program paths.

### 3.2.7 Asynchronous Events: Interrupts and Exceptions

Asynchronous events, such as interrupt requests (IRQs) and processor exceptions, pose a unique challenge to the strict sequential and deterministically chained execution flow described so far. These events occur at unpredictable times, forcing a transfer of control to a specific handler routine outside the program's planned control flow graph. To maintain the security guarantees of cryptographic instruction chaining across asynchronous context switches, the system must securely handle the saving of the interrupted task's state and key context, the execution of the handler, and the secure restoration of the original task's state upon return. When an asynchronous event occurs:

1. *Interrupt Recognition and State Save:* The secure processor recognizes the interrupt or exception. Before executing the first instruction of the handler, it must securely save the full execution state of the interrupted task. This includes:
  - The Instruction Pointer  $ip$  of the instruction that *would have* executed next.
  - The current register state  $R = \{r_1, r_2, \dots, r_n\}$  in their encrypted form.
  - Crucially, the current cryptographic key context, specifically the  $K_{prev}$  and  $K_{next}$  keys associated with the instruction at the saved  $ip$ .

This entire saved state context must be encrypted and authenticated using the master key  $K_M$  and pushed onto a dedicated, secure system stack or

save area. Similar to the function call stack (Section 3.2.5), this secure save area prevents tampering with the interrupted task's context.

$$Saved\_Context = E_{K_M}(ip \parallel R \parallel K_{prev\_saved} \parallel K_{next\_saved} \parallel H_{K_M}(ip \parallel R \parallel K_{prev\_saved} \parallel K_{next\_saved}))$$

2. *Handler Entry Key Derivation:* The system must determine the entry point address of the specific interrupt or exception handler. This lookup process (e.g., via a vector table) is assumed to be secure and protected. The handler's entry point needs a  $K_{prev}$  key to begin execution within the secure environment. This key must be deterministically derived in a way that is secure and tied to the specific handler.

Similar to function calls (Section 3.2.5), we use a deterministic derivation based on the handler's secure identifier or address and the master key  $K_M$ :

$$K_{handler\_entry} = func(handler\_ID) = H_{K_M}(handler\_ID)$$

This  $K_{handler\_entry}$  becomes the  $K_{prev}$  for the first instruction of the handler.

3. *Handler Execution:* The processor transfers control to the handler's entry point using  $K_{handler\_entry}$  as the initial  $K_{prev}$ . The handler executes using the standard sequential chaining mechanism (Section 3.2.2). If the handler involves branches, jumps, or function calls, those are handled as described in Sections 3.2.5–3.2.6, maintaining chain continuity *within* the handler's execution path.
4. *Return from Handler:* A special instruction, analogous to *RET*, is required to return from an interrupt or exception handler (e.g., *RET*). The  $K_{prev}$  for this *RET* instruction is the  $K_{next}$  generated by the last instruction of the handler.

The execution of *RET* involves:

- Retrieving and decrypting the Saved\_Context from the secure save area using  $K_M$ .
- Authenticating the restored context using the included HMAC tag. If verification fails, execution halts.
- Restoring the saved  $ip$ , register state (now decrypted using  $K_{prev\_saved}$ ), and the saved key context  $K_{prev\_saved}$  and  $K_{next\_saved}$ .
- The restored  $K_{next\_saved}$  becomes the  $K_{prev}$  for the instruction at the restored  $ip$ .

## 3.3 Register and Memory Protection

Our secure execution environment implements a protection mechanism for both register content and mem-



ory locations, ensuring confidentiality and integrity throughout program execution.

### 3.3.1 Register Protection

Each register’s contents are encrypted and authenticated after the execution of every instruction. For a set of registers  $R = \{r_1, r_2, \dots, r_n\}$ , the protection scheme operates as follows:

- a) Before instruction execution:
  - Registers are decrypted using  $K_{prev}$  from the current instruction
  - $r_j = D_{K_{prev}}(R[j])$  for each register  $j \in \{1, \dots, n\}$
- b) After instruction execution:
  - Registers are encrypted using  $K_{next}$  from the current instruction
  - For each register  $j$ :

$$R[j] = E_{K_{next}}(r_j || H_{K_{next}}(r_j))$$

### 3.3.2 Memory Protection

Data memory locations are encrypted (with integrity protection) using the master key:

$$M[addr] = E_{K_M}(data || H_{K_M}(data))$$

This allows data sharing between different code sections while maintaining confidentiality and integrity.

## 3.4 System Architecture Overview

The secure execution environment’s components, including the secure processor, encrypted instruction/data memory, encrypted register file, and cryptographic modules (encryption, HMAC, key generator) with Master Key inside the tamper proof region. The architecture enforces a strict in-order execution pipeline where each stage (fetch, decrypt, execute, encrypt) depends on the previous instruction’s cryptographic key.

### Impact on High-Performance Features

- *Pipelining*: Pipeline stages integrate decryption and key derivation. For example, the “Decrypt” stage verifies HMAC tags and decrypts registers using  $K_{prev}$ , while the “Encrypt” stage updates registers with  $K_{next}$ . This introduces latency but preserves pipelining for sequential code.
- *Out-of-Order Execution*: Disabled by design, as the key chain enforces sequential dependencies. Performance loss is mitigated through hardware-accelerated cryptography (e.g., AES-NI).

- *Branch Prediction*: Predictions are allowed, but mispredictions force a pipeline flush. Target labels and derived keys ( $K_{next\_true}$ ) are deterministic, enabling precomputation for predicted paths.

## 4 SECURITY ANALYSIS

Our security analysis follows a systematic catalog-based methodology that examines the system’s defenses across multiple attack surfaces and security properties. The framework evaluates four primary interconnected categories: execution order security, control flow security, state protection, and key management. Execution order security encompasses protections against out-of-order execution, instruction re-ordering resistance, sequential execution guarantees, and edge case handling. Control flow security addresses branch instruction protection, jump target validation, function call/return integrity, and key chain continuity. State protection covers register state confidentiality, memory access security, stack integrity, and context switching security. Key management examines key derivation security, forward/backward secrecy, key chain integrity, and cryptographic binding strength 5.2. This structured methodology enables us to evaluate the system’s security properties and demonstrate resistance against both known attack patterns and potential new threat vectors.

### 4.1 Security Against Out-of-Order Execution

Our implementation practically prevents known attacks like the “Chosen Instruction Attack” from [Rass and Schartner, 2016]. Our design incorporates several features to counter chosen instruction attacks and provide security against Out-of-Order execution:

#### 4.1.1 Key Chaining Protection

- a) *Initial State Security*: The first instruction  $I_0$  establishes security through the master key  $K_M$  as its  $K_{prev}$ , where this master key remains exclusively known to the secure processor. For initialization,  $K_{next}$  is generated randomly for each instruction (except for labels).
- b) *Sequential Execution Protection*: Each subsequent instruction  $I_i$  maintains execution order by requiring its  $K_{prev}$  to match the  $K_{next}$  of the previous instruction. The register decryption process succeeds only when this key matching is valid. The system generates a fresh  $K_{next}$  using a cryp-

tographic RNG, ensuring uniqueness and unpredictability.

- c) Chain Integrity: The instruction integrity remains protected through HMAC validation under the master key. The system establishes cryptographic bindings between consecutive instructions through key transitions. Throughout execution, all register states maintain their encrypted form between instructions, preventing unauthorized access or modification.

#### 4.1.2 Attack Resistance

The system resists attacks by generating unique, unpredictable  $K_{next}$  values via a cryptographic RNG, rendering key prediction negligible; HMAC protection ensures instruction integrity, preventing forgery without the master key; and cryptographic chaining validates each step, making reordering infeasible under standard assumptions.

#### 4.1.3 Sequential Execution Guarantees

The system ensures sequential execution through cryptographic chaining, requiring key validation at each step to maintain secrecy and halt violations; it verifies predecessors via the key chain, protecting register states for confidentiality and integrity, while preventing skipping, replays, and insertions through cryptographic bindings.

#### 4.1.4 Analysis of Key Chain Edge Cases

- a) Branch Point Instructions: When a conditional branch instruction has been executed, both paths must have valid keys, i.e.  $K_{next\_true}$  derived from target label and  $K_{next\_false}$  randomly generated. An attacker might try to force execution down the unintended path. However, this fails since branch condition evaluation depends on register values that remain encrypted and MAC-protected, preventing condition tampering.
- b) Function Entry Points: Multiple CALL instructions targeting the same function could provide an opportunity since they all generate the same  $K_{next}$  through  $func(target\_label)$ . An attacker might try to redirect execution to a different valid call site. However, the call stack's encryption and integrity protection and the protected register state prevent unauthorized control flow changes.
- c) Convergence Points: Program points where multiple execution paths converge, such as after conditional blocks or at function returns, necessarily

share the same next instruction with the associated key. An attacker might attempt to jump to these points prematurely. This fails because:

- The register state remains encrypted under the previous path's keys
- The stack state containing return information is encrypted and integrity-protected
- Key derivation depends on the master key  $K_M$ , which remains inaccessible
- The MAC mechanism protects all state transitions

Therefore, even in cases where multiple paths lead to instructions sharing the same next key, the cryptographic protections on the program state prevent an attacker from successfully exploiting these situations to execute instructions out-of-order.

The system maintains its security guarantees through the combination of encrypted states, protected control flow information, and key derivation chain, even in the presence of multiple valid execution paths sharing key information.

## 4.2 Security Analysis of Control Flow Mechanism

The control flow mechanism's security is built upon several interconnected layers of protection. At its foundation, conditional branching security ensures secure execution through multiple key properties. The system evaluates branch conditions using register values decrypted with the previous key ( $K_{prev}$ ), keeping these decrypted values confined within a secure environment. This evaluation process produces a binary outcome that definitively determines the execution path.

Each branch instruction maintains two distinct next keys:  $K_{next\_true}$  for the branch path taken and  $K_{next\_false}$  for the branch path not taken. The true path key is deterministically derived from the target label using the function  $func(target\_label)$ . This dual-key approach prevents unauthorized path execution while maintaining the integrity of the key chain. The system further protects branch targets through encryption within the instruction itself, with integrity checks preventing any tampering with target addresses. The key derivation mechanism creates a cryptographic binding between each target and its execution path.

### 4.2.1 Unconditional Jump Security

For unconditional jumps, the system maintains security through several mechanisms. Jump targets are

encrypted within the instruction, and integrity verification prevents any target manipulation. The deterministic key derivation ensures consistent and secure target resolution. The key chain remains unbroken across jumps through careful key management, where the next key is derived deterministically from the jump target. This ensures that only authorized code at the target labels can execute.

#### 4.2.2 Function Call and Return Security

The function call mechanism ensures security by encrypting return addresses and pairing them with a random key ( $K_{ret}$ ) on the stack, maintaining key chain continuity across boundaries. Context and register states are encrypted during switches, preventing tampering. Returns verify stack data and  $K_{ret}$ , restoring the context securely to reconnect the key chain, allowing execution only to verified addresses. This robust system prevents return-oriented programming (ROP), jump-oriented programming (JOP), control-flow hijacking, and stack-smashing attacks through cryptographic bindings and integrity checks, leveraging instruction confidentiality and integrity for comprehensive protection.

### 4.3 Security Against Asynchronous Events

The handling of asynchronous events like interrupts and exceptions is secured through a dedicated mechanism built upon the principles of cryptographic state protection and deterministic key derivation, complementing the control flow security discussed in Section 3.2.7.

- **Saved Context Confidentiality and Integrity:** The entire state of the interrupted task, including the instruction pointer, register contents (in their encrypted form), and the crucial key context ( $K_{prev\_saved}, K_{next\_saved}$ ), is encrypted and authenticated using the master key  $K_M$  before being stored in a secure save area. This prevents an adversary from observing or tampering with the task's state or key material while the handler is executing.
- **Secure Handler Entry:** The  $K_{prev}$  key used for the first instruction of the interrupt/exception handler ( $K_{handler\_entry}$ ) is derived deterministically using the master key  $K_M$  and a secure identifier for the specific handler ( $H_{K_M}(handler\_ID)$ ). This prevents an attacker from forcing execution into an unauthorized handler or entering a legitimate handler with a predictable key, as computing the correct entry key requires the master key.

- **Tamper-Evident Return:** The *IRET* instruction verifies the integrity of the restored context (including the saved keys) using the HMAC tag generated under  $K_M$ . Any modification to the saved state between interruption and return will be detected, causing execution to halt.
- **Key Chain Continuity:** The mechanism ensures seamless key chain continuity for the interrupted task upon return. By saving the original  $K_{prev}$  and  $K_{next}$  and making the saved  $K_{next}$  the  $K_{prev}$  for the resumed instruction, the system effectively re-establishes the correct cryptographic link, allowing the task to continue within its original, verified instruction sequence. This prevents attackers from injecting instructions or replaying saved contexts at arbitrary points.
- **Isolation:** The handler executes within its own secure context, starting with a key independent of the interrupted task's chain (derived from the handler ID, not the interrupted task's keys). This provides isolation between the handler and the interrupted task's execution state and key material.

### 4.4 Forward and Backward Secrecy

Informal proof that learning any single instruction-key  $K_i$  does not compromise earlier or later keys.

*Forward Secrecy:* Even if an adversary learns  $K_i$ , they cannot compute  $K_{i+1}, K_{i+2}, \dots$  because each future key is either  $K_{j+1} \leftarrow \text{KeyGen}(t)$  (a fresh, independent RNG output). Neither case allows inversion from  $K_i$  to  $K_{i+1}$ .

*Backward Secrecy:* Even if an adversary learns  $K_i$ , they cannot recover  $K_{i-1}, K_{i-2}, \dots$  because both RNG-drawn keys and PRF-derived keys are pre-image-resistant:  $K_i = \text{KeyGen}(t)$  admit no efficient mapping back to earlier values.

## 5 PERFORMANCE ANALYSIS

This section describes an analysis of the computational overheads introduced by our secure execution environment.

### 5.1 Performance Analysis

The performance analysis considers a program with  $n$  total instructions, including  $n_{cb}$  conditional branch (BEQ) instructions,  $n_c$  function calls (CALL),  $n_r$  function returns (RET),  $n_j$  unconditional jumps (JMP), and  $n_m$  memory operations (LOAD/STORE).

The core cryptographic operations each have complexity  $O(k)$  for input size  $k$ : encryption ( $\mathcal{E}$ ), decryption ( $\mathcal{D}$ ), HMAC computation ( $\mathcal{H}$ ), and key derivation ( $\mathcal{K}$ ).

For basic instruction execution, each instruction requires encryption/decryption of both the instruction itself and the register state ( $\mathcal{E} + \mathcal{D}$ ) each, plus two HMAC operations ( $2\mathcal{H}$ ), yielding a complexity of  $O(n(2\mathcal{E} + 2\mathcal{D} + 2\mathcal{H}))$ , which simplifies to  $O(nk)$ . Control flow operations add additional overhead: conditional branches require  $O(n_{cb}(2\mathcal{K} + \mathcal{E} + \mathcal{D}))$ , function calls need  $O(n_c(2\mathcal{E} + \mathcal{H} + \mathcal{K}))$ , returns take  $O(n_r(\mathcal{D} + \mathcal{H}))$ , and unconditional jumps use  $O(n_j(\mathcal{K} + \mathcal{E}))$ . Memory operations contribute  $O(n_m(\mathcal{E} + \mathcal{D} + \mathcal{H}))$ .

The total computational complexity sums these components:

$$\begin{aligned} C_{total} &= O(n(2\mathcal{E} + 2\mathcal{D} + 2\mathcal{H})) + O(n_{cb}(2\mathcal{K} + \mathcal{E} + \mathcal{D})) + \\ &O(n_c(2\mathcal{E} + \mathcal{H} + \mathcal{K})) + O(n_r(\mathcal{D} + \mathcal{H})) + \\ &O(n_j(\mathcal{K} + \mathcal{E})) + O(n_m(\mathcal{E} + \mathcal{D} + \mathcal{H})) \\ &= O(k(n + n_{cb} + n_c + n_r + n_j + n_m)) \end{aligned}$$

Since  $n_{cb}, n_c, n_r, n_j, n_m$  are all subsets of total instructions  $n$ , their sum cannot exceed  $n$ . Therefore, the final complexity simplifies to  $C_{total} = O(nk)$ .

While our scheme provides strong security guarantees, its strict serialization introduces performance trade-offs compared to modern out-of-order architectures. In out-of-order designs, instruction reordering and speculative execution hide latency and maximize functional unit utilization. By contrast, our cryptographic chaining creates hard dependencies that serialize pipeline stages, leading to:

1. *Pipeline stalls*: Instructions cannot proceed until predecessors are decrypted and verified.
2. *Branch penalties*: Conditional branches exacerbate delays, as both paths require cryptographic validation before resolution.
3. *Limited parallelism*: Functional units remain idle if subsequent instructions are cryptographically blocked.
4. *Asynchronous Event Latency*: Handling asynchronous events (interrupts, exceptions) incurs significant overhead. This includes:
  - The time and computational cost of encrypting and authenticating the entire execution state (registers, IP, and key context) upon interruption.
  - The overhead of pushing this context onto a secure stack/save area.
  - The latency of deriving the handler entry key.

- The cost of retrieving, decrypting, and authenticating the saved context upon returning from the handler.
- A necessary pipeline flush and stall when interrupting and resuming execution.

This overhead is incurred each time an interrupt or exception is processed, adding latency that scales with the amount of state that needs to be saved and restored, impacting the responsiveness of the system to frequent asynchronous events.

These constraints align our model closer to in-order processors, trading performance for security. While our analysis shows linear overhead ( $O(nk)$ ), practical implementations should target security-sensitive workloads where performance degradation is acceptable relative to the threat model.

## 5.2 Fault Injection Simulations

The paper acknowledges the need for empirical validation of fault detection capabilities. While the Python proof-of-concept demonstrates functional correctness, additional simulations will be conducted to evaluate fault resistance. For instance:

- i) **Fault Injection Simulation**: We will model fault injection attacks (e.g., glitches, instruction tampering) to validate detection via HMAC verification and key chain integrity checks.
- ii) **Hardware Faults**: Using RISC-V VP+++ [Schlägl et al., 2024a], we will simulate hardware-level faults (e.g., register/memory corruption) to assess the system's ability to detect and halt execution under tampering. Results will quantify detection rates for various fault models.

## 6 CONCLUSION

This paper has presented a novel secure execution environment that provides comprehensive protection for program execution through a unified cryptographic approach. The system employs authenticated encryption and a key chaining mechanism to ensure instruction confidentiality, integrity, and correct execution ordering across both linear code sequences and complex control flow structures like branches, jumps, and function calls.

At the core of our design is a cryptographic binding between consecutive instructions that prevents reordering and replay attacks while maintaining precise control over execution order. The key chaining mechanism creates cryptographic dependencies between

instructions that enforce their intended sequence. For control flow operations, we introduced deterministic key derivation methods that maintain security properties across branches and function boundaries while supporting legitimate program behavior.

The system employs authenticated encryption and a key chaining mechanism to ensure instruction confidentiality, integrity, and correct execution ordering across both linear code sequences and complex control flow structures like branches, jumps, function calls, and asynchronous events like interrupts and exceptions.

At the core of our design is a cryptographic binding between consecutive instructions that prevents reordering and replay attacks while maintaining precise control over execution order. The key chaining mechanism creates cryptographic dependencies between instructions that enforce their intended sequence. For control flow operations, we introduced deterministic key derivation methods that maintain security properties across branches and function boundaries while supporting legitimate program behavior, extending this to securely manage state and keys across asynchronous interruptions.

Our security analysis demonstrated the system's resistance to various attack vectors including instruction reordering, tampering, and control flow hijacking attempts. The performance analysis revealed that while the system introduces computational overhead from cryptographic operations, it scales linearly with program size ( $O(nk)$  where  $n$  is the number of instructions and  $k$  is the security parameter), making it practical for real-world applications. The successful implementation of a proof-of-concept in Python validated both the theoretical framework and its practical viability. In terms of future work, a natural next step is to apply and evaluate the concepts presented in this paper on real-world architectures. A suitable candidate is the relatively young RISC-V Instruction Set Architecture (ISA) [Waterman et al., 2011, Waterman et al., 2016], as it is an open standard. Efficient design space exploration and evaluation can be achieved using simulation techniques such as SystemC/TLM (IEEE 1666 [noa, 2023]) based virtual prototyping [De Schutter, 2014, Leupers et al., 2012]. The very capable open source RISC-V VP++ [Schlägl et al., 2024a, Schlägl et al., 2024b, Schlägl and Große, 2025] is, being considered for this. A promising direction for future work is exploring hybrid architectures where cryptographic constraints apply selectively to security-critical code regions (e.g., cryptographic routines or authentication logic), while non-critical code executes on traditional out-of-order pipelines. Such an approach could balance performance and secu-

rity, leveraging hardware/software co-design to isolate sensitive computations within cryptographically enforced boundaries.

## Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria. We thank Dr. Zahra Seyedi for drawing the flowchart picture for this paper.

## REFERENCES

- (2023). IEEE Standard for Standard SystemC® Language Reference Manual. *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)*, pages 1–618. Conference Name: IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011).
- Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2009). Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1).
- Blazy, S. and Trieu, A. (2016). Formal verification of control-flow graph flattening. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016*, pages 176–187. New York, NY, USA. Association for Computing Machinery.
- Chamelot, T., Couroussé, D., and Heydemann, K. (2022). Sci-fi: control signal, code, and control flow integrity against fault injection attacks. In *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe, DATE '22*, page 556–559, Leuven, BEL. European Design and Automation Association.
- Christou, G., Vasiliadis, G., Papaefstathiou, V., Papadogiannakis, A., and Ioannidis, S. (2020). On Architectural Support for Instruction Set Randomization. *ACM Trans. Archit. Code Optim.*, 17(4):36:1–36:26.
- Costan, V. and Devadas, S. (2016). Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86.
- de Clercq, R., Götzfried, J., Übler, D., Maene, P., and Verbauwhede, I. (2017). SOFIA: Software and control flow integrity architecture. *Computers & Security*, 68:16–35.
- De Schutter, T. (2014). *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press.
- Ege, B., Kavun, E. B., and Yalçın, T. (2011). Memory Encryption for Smart Cards. In *Smart Card Research and Advanced Applications*, pages 199–216, Berlin, Heidelberg. Springer.
- Gallagher, M., Biernacki, L., Chen, S., Aweke, Z. B., Yitbarek, S. F., Aga, M. T., Harris, A., Xu, Z., Kasikci, B., Bertacco, V., Malik, S., Tiwari, M., and Austin, T. (2019). Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proceedings of the*

- Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, pages 469–484, New York, NY, USA. Association for Computing Machinery.
- Gruss, D., Spreitzer, R., and Mangard, S. (2015). Cache template attacks: automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Conference on Security Symposium, SEC'15*, page 897–912, USA. USENIX Association.
- Gueron, S. (2016). A memory encryption engine suitable for general purpose processors. *Cryptology ePrint Archive*, Paper 2016/204.
- Harris, A., Verma, T., Wei, S., Biernacki, L., Kisil, A., Aga, M. T., Bertacco, V., Kasikci, B., Tiwari, M., and Austin, T. (2021). Morpheus II: A RISC-V Security Extension for Protecting Vulnerable Software and Hardware. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 226–238.
- Henson, M. and Taylor, S. (2014). Memory encryption: A survey of existing techniques. *ACM Comput. Surv.*, 46(4):53:1–53:26.
- Kc, G. S., Keromytis, A. D., and Prevelakis, V. (2003). Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security, CCS '03*, pages 272–280, New York, NY, USA. Association for Computing Machinery.
- Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2018). Spectre Attacks: Exploiting Speculative Execution. arXiv:1801.01203.
- Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2020). Spectre attacks: exploiting speculative execution. *Commun. ACM*, 63(7):93–101.
- Leupers, R., Martin, G., Plyaskin, R., Herkersdorf, A., Schirrmeister, F., Kogel, T., and Vaupel, M. (2012). Virtual platforms: Breaking new grounds. In *Design, Automation and Test in Europe*, pages 685–690.
- Liu, C., Wang, X. S., Nayak, K., Huang, Y., and Shi, E. (2015). OblivM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. ISSN: 2375-1207.
- Maas, M., Love, E., Stefanov, E., Tiwari, M., Shi, E., Asanovic, K., Kubiatiowicz, J., and Song, D. (2013). PHANTOM: Practical Oblivious Computation in a Secure Processor. *CCS '13*, pages 311–324. ACM.
- Rass, S. and Schartner, P. (2016). On the Security of a Universal Cryptocomputer: the Chosen Instruction Attack. *IEEE Access*, 4:7874–7882. Conference Name: IEEE Access.
- Rass, S., Schartner, P., and Wamser, M. (2015). Oblivious Lookup Tables. *Tatra Mountains Mathematical Publications*, 67.
- Schlägl, M. and Große, D. (2025). Fast interpreter-based instruction set simulation for virtual prototypes. In *Design, Automation and Test in Europe Conference (DATE)*.
- Schlägl, M., Hazott, C., and Große, D. (2024a). RISC-V VP++: Next generation open-source virtual prototype. In *Workshop on Open-Source Design Automation*.
- Schlägl, M., Stockinger, M., and Große, D. (2024b). A RISC-V “V” VP: Unlocking Vector Processing for Evaluation at the System Level. In *Design, Automation and Test in Europe Conference (DATE)*, pages 1–6.
- Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security, CCS '07*, pages 552–561, New York, NY, USA. Association for Computing Machinery.
- Stecklina, O., Langendörfer, P., Vater, F., Kranz, T., and Leander, G. (2015). Intrinsic Code Attestation by Instruction Chaining for Embedded Devices. In *Security and Privacy in Communication Networks*, pages 97–115, Cham. Springer International Publishing.
- Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. (2014). Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA. USENIX Association.
- Tsoutsos, N. G. and Maniatakos, M. (2014). HEROIC: Homomorphically Encrypted One Instruction Computer. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. ISSN: 1558-1101.
- Wang, X., Chen, H., Jia, Z., Zeldovich, N., and Kaashoek, M. F. (2012). Improving integer security for systems with KINT. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 163–177, Hollywood, CA. USENIX Association.
- Waterman, A., Lee, Y., Avizienis, R., Patterson, D. A., and Asanovic, K. (2016). The risc-v instruction set manual volume ii: Privileged architecture version 1.7. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129*.
- Waterman, A., Lee, Y., Patterson, D. A., and Asanovic, K. (2011). The risc-v instruction set manual, volume i: Base user-level isa. *eeecs department. University of California*.
- Werner, M., Unterluggauer, T., Schaffenrath, D., and Mangard, S. (2018). Sponge-Based Control-Flow Protection for IoT Devices: 2018 IEEE European Symposium on Security and Privacy. *2018 IEEE European Symposium on Security and Privacy*. Publisher: IEEE.
- Zahur, S. and Evans, D. (2015). Obliv-c: A language for extensible data-oblivious computation. *Cryptology ePrint Archive*, Paper 2015/1153.
- Zahur, S., Rosulek, M., and Evans, D. (2015). Two Halves Make a Whole. In *Advances in Cryptology - EUROCRYPT 2015*, pages 220–250, Berlin, Heidelberg. Springer.