Code Encryption with Intel TME-MK for Control-Flow Enforcement

Martin Unterguggenberger, Lukas Lamster, Mathias Oberhuber, Simon Scherer, and Stefan Mangard

Graz University of Technology, Austria

Abstract. Memory safety errors enable an adversary to corrupt code pointers, diverting the program's control flow. Recent CPU features, such as Intel CET/IBT, harden software systems against exploitation attempts that maliciously redirect control flow operations. While IBT limits valid indirect branch targets, forward-edge transfers can still be redirected to any IBT-marked function. Thus, IBT cannot provide fine-grained protection against forward-edge control-flow attacks.
This paper presents code encryption with Intel TME-MK, a novel approach for control-flow enforcement against software exploitation on off-the-shelf x86 machines. We repurpose the Intel TME-MK runtime encryption to achieve function-level code encryption. Encrypted functions are only accessible through function pointers associated with the correct key, thereby enforcing fine-grained restrictions for control-flow transfers. We demonstrate two new encryption-based techniques for software hardening in practice: forward-edge control-flow integrity and library encryption. We implement a security-hardened toolchain that combines compiler instrumentation and a loader extension to ensure the validity of the program's execution flow through efficient hardware-backed encryption. Our prototype shows a geomean performance overhead of 7.8 % for forward-edge control-flow integrity and 2.2 % for library encryption evaluated with the SPEC CPU2017 benchmark suite.

Keywords: Code Encryption · Control-Flow Integrity · Intel TME-MK.

1 Introduction

Coding errors that introduce memory safety bugs into software systems are a severe threat to platform security [45, 49, 63]. Such memory errors are a main root cause for enabling critical zero-day exploits [22,25,50]. A malicious actor can exploit how (non-memory safe) C/C++ software interacts with memory [53,61]. This way, an adversary can overwrite a code pointer, e.g., a function pointer or a return address, to divert the program's execution flow. Control-flow redirection allows for advanced exploitation techniques that are generally classified as code-reuse attacks. For instance, return-into-libc [59] overwrites a return address to redirect the control flow to a library function (e.g., `system` in libc). Furthermore, ROP [11, 54] and COP/JOP [9, 15] are powerful attacks that allow for arbitrary attack flows by chaining existing code gadgets located in executable memory.

Software-hardening defenses, such as security-enhancing compiler options, are essential for exploit mitigation throughout the entire low-level software stack. Security extensions need to harden legacy and future C/C++ codebases while being constrained to strict system requirements, e.g., minor performance impact and compatibility with legacy x86 software. To maintain efficiency and compatibility while preventing the exploitation of vulnerabilities, defense mechanisms must be integrated into the processor and enforced in hardware [38, 51, 60, 66].

CPU features, such as Intel Control-flow Enforcement Technology (CET) [60], address specific sub-classes of control-flow hijacking attacks. Intel CET provides a shadow stack feature that ensures the integrity of return addresses. This strong protection against return address manipulation enables backward-edge control-flow integrity (CFI) [1, 12, 14]. Moreover, Intel CET introduces the Indirect Branch Tracking (IBT) feature, which explicitly marks valid destinations for the program's execution flow. Thus, forward-edge control-flow transfers, such as indirect jumps and calls, are exclusively limited to IBT-marked functions, which are identified via designated landing pad instructions. Any indirect branch to an unmarked address is interpreted as a control-flow hijacking attempt and causes an exception. However, IBT can only provide coarse-grained CFI. As all valid targets are marked using the same landing pad instruction, IBT cannot distinguish between multiple valid indirect branch targets. Thus, while IBT makes exploitation harder, this imprecision still allows an attacker to divert the control flow to any IBT-marked location anywhere in the program's code region [14, 56].

In this paper, we present a novel approach for control-flow enforcement through fine-grained code encryption on off-the-shelf x86 machines. We repurpose Intel's encryption technology, Total Memory Encryption Multi-Key (TME-MK) [27, 29], for function-granular code encryption. Intel TME-MK implements efficient runtime encryption, which is typically used for page-granular encryption of confidential virtual machines (VM) [28, 30]. Our design builds on top of the TME-MK feature to encrypt code on a function granularity through the use of page aliasing, a technique that allows multiple virtual addresses to refer to the same physical memory. Function-granular code encryption enables two new toolchain hardening techniques: forward-edge CFI and library encryption.

Code encryption and linking function pointers with encryption keys restrict access to application or library functions. This approach limits available functions within the program, thereby mitigating control-flow attacks by enforcing fine-grained CFI policies or preventing the code reuse of security-critical library functions. The forced use of encryption keys for function pointers restricts control-flow transfers, as incorrect decryption leads to garbled code and, thus, results in the execution of arbitrary pseudo-random instructions, which likely causes a fault [33]. Compared to IBT, which limits indirect branches to any marked function entry, our design restricts the overall set of accessible functions by leveraging up to 32K encryption keys. Moreover, combining our code encryption with IBT offers synergies, *i.e.*, confining indirect branches to the function entry and reducing the set of accessible functions. Also, violations are detected, as the incorrect decryption of code is unlikely to lead to valid landing pads.

We implement a hardened toolchain consisting of an LLVM [37] extension, a modified loader, and a kernel patch, enabling control-flow enforcement through compiler instrumentation with hardware-backed encryption. Moreover, we provide an in-depth security analysis of our design and a performance evaluation using SPEC CPU2017 [10]. Our prototype shows practical results with a geomean overhead of 7.8 % for forward-edge CFI and 2.2 % for library encryption.

Contributions. In summary, we make the following contributions:

1. We present code encryption with Intel TME-MK, a novel approach that enforces fine-grained security policies on control-flow transfers by encrypting functions and linking function pointers to encryption keys.
2. We provide new insights on the application of code encryption: We detail two software hardening techniques, *i.e.*, forward-edge CFI and library encryption. In addition, we outline synergies of our encryption approach, resulting in garbled code, with detection through wrongly decrypted IBT landing pads.
3. We develop a prototype of our security-hardened toolchain and evaluate the performance, showcasing practical results for SPEC CPU2017.
4. We conduct an in-depth security analysis, highlighting the security properties and efficacy of our encryption-based design.

Outline. The paper is organized as follows. Section 2 provides the background of this work. Section 3 defines our threat model. Section 4 presents the design, and Section 5 describes the implementation of our prototype. Section 6 and Section 7 provide the security analysis and performance evaluation. Section 8 discusses related and future work. Section 9 concludes this work.

2 Background

This section provides the background on control-flow hijacking attacks, Intel Control-flow Enforcement Technology (CET), and Intel Total Memory Encryption Multi-Key (TME-MK).

2.1 Control-Flow Attacks

Software that is developed in non-memory safe programming languages (e.g., C and C++) is vulnerable to memory safety errors. These memory safety vulnerabilities, introduced by coding errors, enable a malicious actor to corrupt data located in memory. Consequently, this also allows an adversary to modify code pointers (e.g., function pointers or return addresses) through illegitimate memory interactions. These code pointers are then used by control flow instructions (e.g., `ret`, `call`, or `jmp` instructions). Hence, the corruption of the target address allows the redirection of the program's execution, hijacking the control flow.

In addition, more advanced exploitation techniques, such as return-oriented programming (ROP) [11, 16], aim to reuse existing code from the executable memory to achieve arbitrary attack flows. Specifically, ROP attacks chain together multiple instruction sequences, called ROP gadgets, to achieve arbitrary

attack flows through the exploitation of return instructions. Note that the return instruction retrieves the address of the next instruction from the stack and resumes the program execution from that address. By carefully chaining together these gadgets, also referred to as crafting an ROP chain, and manipulating return addresses located on the stack, the attacker can execute a sequence of gadgets to achieve arbitrary code execution flows.

Similarly, call/jmp-oriented programming (COP/JOP) [9, 15] reuses existing code in memory through the redirection of indirect calls and jumps. For instance, these techniques achieve an arbitrary attack flow through the help of a dispatcher function [9]. Due to their use of gadgets in executable memory, ROP and COP/JOP attacks are generally classified as so-called code reuse attacks.

2.2 Intel Control-Flow Enforcement Technology

Intel Control-flow Enforcement Technology (CET) [60] is a set of architectural elements developed to help ensure the integrity of control-flow transfers within a program. Thereby, the processor is extended with capabilities to enforce control-flow integrity (CFI) [1,12,14] for both forward-edge and backward-edge transfers.

First, Intel CET provides a hardware-based shadow stack feature that offers strong protection against return address modification, thereby providing backward-edge CFI. Specifically, the shadow stack organizes and manages a separate stack that exclusively contains return addresses. On each function call, a copy of the return address is stored on the shadow stack, thus becoming inaccessible to the adversary. When exiting the called function, the return address is taken from the regular stack. In addition, this (potentially modified) address is then compared to the return address stored on the shadow stack. A mismatch during this comparison indicates a corrupted return address and results in an exception. Thus, return-based code-reuse attacks, e.g., return-into-libc [59] and ROP [11, 54] attacks, are detected and mitigated.

Second, Intel CET provides forward-edge CFI through the integration of Indirect Branch Tracking (IBT). IBT extends the x86 ISA with landing pad instructions (e.g., `enbr64`). Typically, a compiler inserts landing pads in function entries to mark valid indirect call/jump targets in executable memory. Since IBT reduces the potential destinations of indirect branches to valid landing pad instructions, it greatly reduces the attack surface for control-flow hijacking attacks such as COP/JOP [9,15]. However, IBT can only provide coarse-grained CFI. As all valid function entries are identified using the same landing pad instructions, an attacker can still divert the control flow to all function entries marked with IBT within the application or (shared) libraries [56].

2.3 Intel Total Memory Encryption Multi-Key

Intel's memory encryption technology, Total Memory Encryption (TME) [27], allows transparent encryption of the system's entire physical memory with a single encryption key. The Total Memory Encryption Multi-Key (TME-MK) [27] extension provides DRAM encryption with multiple encryption keys, enabling

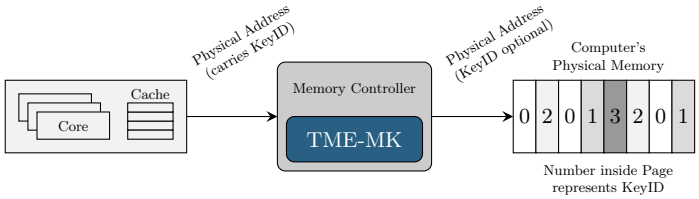

Fig. 1: Overview of the Intel TME-MK memory encryption.

the selection of page-granular encryption keys through the processor page tables [27, 29]. Intel TME-MK is an architectural element mainly used for the encryption of virtual machines (VMs) and containers, thereby ensuring the confidentiality of DRAM data and helping to counteract physical attacks [26, 28].

Figure 1 shows an overview of Intel TME-MK's memory encryption. TME-MK operates transparently on memory transactions between the CPU core and the DRAM memory controller. Writing to memory encrypts the data, and subsequently, reading from memory decrypts the previously encrypted data. TME-MK organizes its cryptographic key material using a key table that maps the key identifiers (keyIDs) to their respective encryption keys. To offer more flexibility, TME-MK supports different encryption modes, such as 128-bit and 256-bit AES-XTS [20, 21, 43, 55] encryption. Additionally, the Intel Trust Domain Extensions (TDX) [28, 30] add support for authenticated encryption with cryptographic integrity through a message authentication code (MAC).

Memory pages are encrypted depending on the keyID encoded into the upper part of the physical address of the memory request. Thus, the physical address carries the keyID to the encryption engine in the memory controller, controlling the encryption key and mode used for the memory interactions. Note that TME-MK is specified for up to 2^{15} encryption keys [27]. As the encoding of keyIDs results in a reduction of addressable physical memory, the size of the keyID is platform-dependent and varies across processors with TME-MK support.

3 Threat Model

We consider an attacker that intends to exploit a memory safety vulnerability to corrupt a code pointer (e.g., a function pointer located in memory), hijacking the control flow of an unprivileged user space program. Thereby, the adversary exploits a vulnerability in an attempt to modify the program state or behavior through the redirection of the program's execution flow. Moreover, we assume that the attacker knows the address space layout of the target program, *i.e.*, the attacker knows the addresses of potentially lucrative branch targets.

Intel CET [60] and comparable security features from other CPU vendors (e.g., AMD Shadow Stack [4] and ARM Guarded Control Stack [5]) are widely

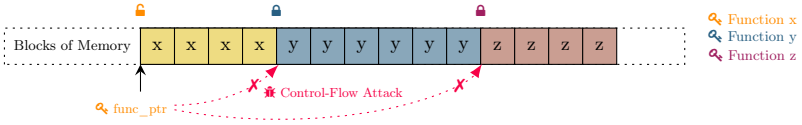

Fig. 2: High-level concept of the function-granular code encryption.

available. Thus, we assume that the Intel CET shadow stack feature is enabled and provides us with backward-edge control-flow integrity.

We assume that the privileged operating system/hypervisor is benign and that writable memory is marked as non-executable (see Write-XOR-Execute). We consider other attack vectors, such as side-channel attacks [35, 41] and fault injection attacks [34, 46, 62], to be out of the scope of this work.

4 Design

In this section, we present our novel technique for control-flow enforcement through code encryption that effectively hardens software against control-flow hijacking attacks. We repurpose the Intel TME-MK encryption engine, available on off-the-shelf Intel x86 CPUs, to encrypt individual functions for the fine-grained restriction of forward-edge control-flow transfers.

4.1 High-Level Overview

At its core, our design encrypts individual functions with designated encryption keys. Encrypted functions are only available for call sites with a matching key that correctly decrypts the function. When performing an indirect call, the key associated with the functions pointer is used to decrypt the call target. Only functions encrypted with the associated key will be decrypted into meaningful code. Thus, our design enables fine-grained control-flow enforcement through code encryption. We repurpose the Intel TME-MK hardware feature to achieve function-granular code encryption on commodity x86 CPUs. Moreover, our hardened toolchain identifies function pointers and applies compiler-based code instrumentation *to enforce the use of designated encryption keys*, depending on the defined security policy. Note that our generic code encryption scheme enables a variety of security policies based on the underlying code encryption mechanism. While this work focuses on a function signature-based policy as a proof-of-concept, other CFI policies [42, 64, 67] can also be implemented.

Figure 2 illustrates a high-level overview of the function-granular code encryption employed by our design. In the example, individual 16 B memory blocks of the code section are encrypted with the different encryption keys assigned to the functions `x`, `y`, and `z`. Moreover, control-flow transfers are instrumented to enforce the usage of a dedicated encryption key defined by the security policy. The example shows a function pointer associated with the encryption key of

function `x` that is dereferenced by an indirect function call. Enforcing the usage of the encryption key for function `x` limits the valid call targets to destinations encrypted with the respective key. This way, any redirection of the control flow by tampering with the function pointer, e.g., to the functions `y` or `z`, leads to a decryption with a wrong key. Decrypting code with an incorrect key leads to garbled code due to a pseudo-random decryption result, which the attacker cannot control. Hence, executing instructions of a function encrypted with a different key is impossible as the attacker can only receive garbled code.

Furthermore, we assume that backward-edge control-flow transfers are protected by Intel CET's shadow stack feature. Note that code encryption also synergizes with the Intel IBT landing pads. Particularly, our encryption approach results in garbled code that is then detected through IBT, as the incorrect decryption of code is very unlikely to produce a valid landing pad instruction.

4.2 Code Encryption with Intel TME-MK

Our design repurposes the Intel TME-MK feature to efficiently encrypt executable code in memory. Intel TME-MK, originally intended for the encryption of entire virtual machines, enables page-granular encryption of memory. The encryption uses up to 15-bit keyIDs encoded in the physical address field of the page table entry to select up to 32K encryption keys [27]. All memory transactions between the CPU core and the main memory (*i.e.*, the DRAM) are transparently encrypted using the selected key. Our design advances this page-level encryption and allows the assignment of different encryption keys to individual functions.

Function-Granular Encryption. While TME-MK operates on page granularity, we can achieve function-granular encryption through page aliasing [57,58,65]. Figure 3 illustrates the function-granular code encryption with Intel TME-MK. Aliasing allows multiple virtual addresses to refer to the same physical memory and is typically used for shared memory. For every keyID, the program's code region is mapped into the virtual address space using a different virtual base address. However, all mapped regions reference the same physical memory. Thus, each keyID has a unique alias that maps to the physical memory using the associated keyID. Note that the alias mappings are chosen so that the code regions do not overlap with each other or with other regions in the virtual address space. At program startup, each function in the code is encrypted by writing it to memory using the corresponding alias for the intended keyID. As the setup of mappings and encryption of individual functions must precede the regular program execution, these steps are performed by the loader. After initialization, the loader sets the permissions of code pages back to *read-only* and *executable*. While the loader ensures that each function is encrypted with the intended keyID, we must also ensure that the program uses the correct keyID for function calls at runtime. We achieve this through compiler instrumentation that forces indirect branches to use the virtual base address of the designated alias mappings and, thus, the corresponding encryption keys for forward-edge control-flow transfers.

The example given in Figure 3 shows a 4 kB code page containing three distinct functions: `x`, `y`, and `z`. Memory aliasing creates different views on the

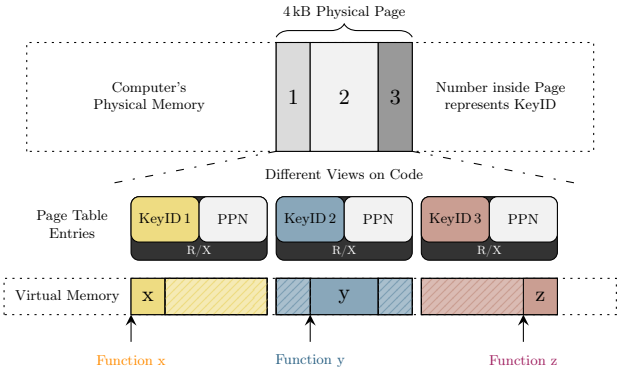

Fig. 3: Overview of the function-granular code encryption through Intel TME-MK. Aliasing creates different views on the computer's physical memory. Here, three functions, `x`, `y`, and `z`, are located on the same physical page while each function is encrypted with its respective encryption key (*i.e.*, keyID 1, 2, and 3). The compiler enforces the usage of keyIDs; thus, code can only be executed through function pointers associated with the correct keyID. This limits control-flow transfers as access with an incorrect keyID results in garbled code.

program's code region, *i.e.*, it allows multiple virtual addresses to refer to the same physical memory location using different keyIDs. The functions `x`, `y`, and `z` can be accessed through the three mappings with the corresponding keyIDs 1, 2, and 3, respectively. Thus, function `x` can only be called when accessed through keyID 1 since other aliases result in garbled code and crash on execution.

The compiler enforces the usage of a specific keyID associated with the function pointer, depending on the security policy. Precisely, our design instruments the virtual addresses of the function pointers to refer to the alias mapping of the code section associated with their designated keyIDs. Depending on the security policy, the control-flow enforcement of our mechanism provides toolchain hardening for forward-edge CFI or library code encryption.

4.3 Control-Flow Enforcement

We leverage our code encryption approach to harden C/C++ software. First, our design allows the enforcement of fine-grained CFI policies to protect forward-edge control-flow transfers. Second, our design enables the encryption of library code that helps to protect against code reuse of security-critical library functions.

Forward-Edge Control-Flow Integrity. Control-flow attacks allow a malicious actor to redirect program execution to different destinations. When an adversary manages to corrupt the address of a function pointer used by a control-flow operation, they can divert the control flow to arbitrary code locations.

To counteract this threat, our design confines indirect branches to individual functions. We achieve this restriction of valid destinations for control-flow trans-

fers by associating keyIDs (that correspond to a specific encryption key) with function pointers. This limits potential branch targets, as functions can only be executed through the use of a function pointer with the correct keyID. In general, our generic design can enforce a variety of CFI policies, e.g., type-based CFI [64]. For our proof-of-concept, we derive the CFI policy from the *function signature* of the target destination. Therefore, the compiler toolchain identifies function pointers and derives the keyIDs from their function signature. The mapping from function signatures to keyIDs is used to associate function pointers with encryption keys through compiler instrumentation. Subsequently, the function signatures and corresponding keyIDs are encoded as metadata into the ELF binary. The modified loader relies on this metadata and is responsible for creating the mappings for all available keyIDs of the code region, and encrypting the functions with the associated encryption keys upon program startup.

Even with Intel IBT, the state-of-the-art for CFI in commodity systems, an attacker can divert the control flow to any marked function entry. Our code encryption technique provides more fine-grained CFI, as forward-edge control-flow transfers only result in meaningful code if the target is decrypted with the intended key. Further, forward-edge CFI within libraries can be easily achieved by statically compiling and linking the application and libraries with our toolchain. In the case of our proof-of-concept CFI policy, attackers are confined to functions of the same function signature, *i.e.*, function signature-granular CFI. For other functions, CFI violations result in garbled code. In such a case, the attacker would force the program to execute arbitrary pseudo-random instructions that likely crash the program [33]. We can additionally improve the reliability by combining our encryption-based approach with IBT. As function entries must be marked by landing pads, most CFI violations are detected by IBT as the incorrectly decrypted code is highly unlikely to result in a valid landing pad. Alternatively, our code encryption approach can use Intel TME-MK with integrity support, detecting incorrect code accesses through a cryptographic MAC.

Library Code Encryption. The redirection of control-flow transfers can also be misused to call or jump into library functions. Library functions are a welcoming exploitation target as they often contain security-critical code, e.g., code reuse of functions in the C standard library [59].

To help protect against code reuse attacks of library functions, we restrict control-flow transfers from the application into shared libraries through security policies. Similar to our CFI scheme discussed above, we instrument the control-flow operations of the application to use a designated encryption key. Here, the application's code is assigned a default encryption key. We allow the selective hardening of software libraries to restrict access to a subset of functions that execute security-critical code. Security-critical library functions, such as `system` or `exec`, are encrypted with dedicated encryption keys that differ from the default key. Thus, invalid paths in the control-flow graph targeting critical library functions are restricted, as the matching key is required for correct decryption.

Library functions that are invoked via indirect branches are confined as the corresponding function pointer is associated with an encryption key due to our

instrumentation. This encryption key defines which functions within the library can be accessed. Moreover, library calls are also instrumented, enforcing the usage of the associated encryption key for the library function. Note that shared libraries also need to be accessible for unprotected programs. Thus, our toolchain uses dedicated versions of the hardened shared libraries for our code encryption to maintain compatibility with unprotected programs.

5 Implementation

In the following, we describe the prototype implementation of our toolchain, which consists of an LLVM extension, a modified loader, and a kernel patch.

5.1 Compiler Extension

We implement our code encryption scheme as an extension to the LLVM [37] compiler (version 17.0.0). Our prototype integrates a set of compiler passes to the LLVM optimizer and the x86 backend. The backend passes insert a custom ELF section for metadata and perform the code instrumentation. Specifically, we implement a compiler pass that creates a custom ELF section and encodes metadata for all functions depending on the used security policy. For forward-edge CFI, our prototype derives the keyID from the function signature by generating a truncated hash value of the LLVM IR type information that maps to a dedicated keyID. For library encryption, the compiler pass allows the selection of keyIDs for individual library functions that are considered security critical.

Moreover, we implement a compiler pass that retrieves the keyIDs and instruments the corresponding control-flow instructions, linking them to the correct alias mappings. To achieve this, we analyze and identify all indirect function calls and instrument the target address by setting its alias bits associated with the above-identified keyID; e.g., the corresponding function signature. Concretely, we implement this by manipulating a specific bit range of the virtual address according to the keyID, depending on the security policy. For indirect branches, the function pointer is instrumented through logical operations, *i.e.*, by clearing the bit range and subsequently inserting the keyID into the alias bit range. This procedure forces the function pointers to use their intended virtual address alias corresponding to the policies' designated encryption key. Note that our compiler framework is fully parameterizable in regard to the bit range of the virtual address space to achieve compatibility with different memory layouts. Therefore, our extension offers two compiler options to define the number of available keyIDs and the bit range of the aliased code mappings. The compiler pass also needs to correct the target addresses for direct function calls within the program. As all functions are encrypted with their associated encryption keys to enforce our CFI policies, direct function calls also need to use the intended virtual address alias (e.g., keyID derived by the function signature) to decrypt functions correctly. Therefore, we patch the content of the `rip` register to use the intended virtual address to access the function with the correct keyID.

We also implement optimization passes operating on the LLVM IR. We need to handle statically allocated data that contain a function pointer in their initializer value, such as function pointers stored in global/static variables. Thus, we invoke a compiler-generated startup routine that correctly initializes global data containing function pointers with the correct keyID. In addition, we assign all static functions a unique function name (being able to differentiate functions with the same name but different signatures) and handle non-compatible operations like comparisons of function pointers. To optimize the performance of forward-edge CFI, our compiler toolchain additionally aligns and pads functions to the cache line size of 64 B. This alignment is essential, as accessing an already cached memory region using a different memory alias forces the hardware to evict the currently cached version [2]. The LLVM compiler already provides an interface for this function alignment. In addition, we use the LLVM support for Intel IBT to insert landing pads for function entries.

For library code encryption, control-flow operations of the program are instrumented similarly to the CFI policy. Here, the compiler either enforces the use of the default encryption key (*i.e.*, keyID 0) or a designated encryption key associated with a security-critical function. Special care is required for shared libraries as these function calls are performed through the PLT. We also map the shared library with all available keyIDs and separately encrypt security-critical functions according to the security policy. The PLT stub uses the GOT entry to either call the dynamic linker to resolve the destination, or already contains the address of the library function. We instrument the PLT stub to enforce the use of the keyID associated with the defined security policy. This way, security-critical functions can only be called from their respective PLT entry or function pointers that are allowed to target this library function.

Note that we use an Intel Xeon Gold 6530 processor as our main development and evaluation platform. While Intel TME-MK supports up to 15-bit keyIDs, our processor supports 6-bit keyIDs, resulting in a total of 64 encryption keys. For our prototype, this means that keyID collisions for some function signatures can occur, which is a limitation of our evaluation platform. However, our compiler extension can also be configured in regards to the supported number of keyIDs of the processor, allowing the use of up to 15-bit keyIDs for future processors.

5.2 ELF Loader

We provide toolchain support for our code encryption scheme with a modified ELF loader. The loader is responsible for creating alias mappings to the code section for all available keyIDs (*i.e.*, 6-bit keyIDs on our evaluation platform). This allows access to the code section through all the associated encryption keys.

Moreover, the loader needs to encrypt individual functions depending on the used security policy. Therefore, the loader relies on the metadata encoded in our custom ELF section previously inserted by the compiler. The loader parses this metadata and encrypts the code of individual functions depending on the security policy, *i.e.*, by writing the function's code with the designated keyID to memory. During this initial setup, we flush cache lines when updating the

associated keyID to ensure all writes are done with a single active keyID, as recommended by the TME-MK specification [27]. After code encryption, we change the page permissions of the code section back to *read-only* and *executable*.

In addition, the loader maps the PLT/GOT for every aliased code section. This is required since PLT calls are performed through instruction pointer relative-addressing. For our prototype, the loader also initializes shared libraries and maps them for all available keyIDs. However, dynamic linker support could be added to achieve an on-demand mapping and encryption of shared libraries.

5.3 Linux Kernel Patch

Operating system support is necessary to provide a software interface to control the Intel TME-MK memory encryption. Intel TME-MK repurposes up to 15-bit of the physical address (starting with the highest order bit available) located in the PTE to encode the keyID. We use an experimental Linux kernel patch provided by Intel Labs that allows to assign keyIDs through a syscall interface [31]. Specifically, the kernel patch enables additional arguments for the `mprotect` system call to associate a keyID with a specific range of memory pages.

6 Security Analysis

This section analyzes the derived security properties of our encryption-based design. We assume an adversary that manipulates a code pointer to redirect the program's control flow (see Section 3). We further distinguish between control-flow attacks that target forward-edge and backward-edge control-flow transfers. For backward-edge control-flow transfers, we assume that the Intel CET shadow stack efficiently protects the return address. Thus, corrupting the return addresses becomes infeasible, *i.e.*, the attacker cannot overwrite return addresses on the shadow stack. Furthermore, the adversary can attempt to gain control over a function pointer to divert the forward-edge control-flow transfer. Here, the attacker exploits a memory safety vulnerability to corrupt a function pointer and hijack the control flow, e.g., a buffer overflow that allows to overwrite and manipulate the function pointer and function arguments.

Function Pointer. Our generic code encryption design ensures fine-grained control-flow enforcement by linking function pointers with encryption keys. If an adversary gains control of a function pointer and manipulates its address to redirect control flow, the attack surface is confined. The compiler ensures that function pointers are instrumented to use their designated keyID, restricting forward-edge control-flow transfers solely to destinations that are permitted by the defined security policy. Furthermore, keyID forgery is prevented through our code instrumentation since the correct alias bits are explicitly set depending on the security policy directly before dereferencing the function pointer.

CFI Policy. Our generic design enables the implementation of different policies, such as CFI and library encryption. The prototype implements a CFI policy that derives valid indirect control-flow targets from *function signatures*. This

signature-based CFI policy associates a designated keyID (mapping to an encryption key) for each function signature. This way, function pointers are limited to solely target destinations with their intended function signature, *i.e.*, signature-granular CFI confining the control flow to matching signatures. Note, however, that other CFI policies [42, 64, 67] can also be implemented on top of our code encryption scheme. In theory, our design can take advantage of up to 32K encryption keys, as Intel TME-MK is specified for up to 15-bit keyIDs, representing the total number of distinct encrypted functions leveraged by the CFI policy. Nevertheless, the number of available keyIDs is platform-specific, e.g., our evaluation platform supports 6-bit keyIDs. This can result in keyID collisions for some function signatures, which is a limitation of our evaluation platform.

Code Encryption. Our design enforces that program execution receives code decrypted with the encryption key that corresponds to the used keyID. For policy violations, this results in the (incorrect) decryption of garbled code and, subsequently, the execution of arbitrary pseudo-random instructions. Precisely, the use of the wrong keyID leads to a cache miss and results in the data being served from DRAM, where the TME-MK encryption engine decrypts the data. The use of an incorrect keyID causes the first instruction fetch to be decrypted with the wrong encryption key, executing garbled code and very likely causing a fault [33]. Moreover, our design uses IBT landing pads to detect control-flow attacks violating our security policies. It is highly unlikely that garbled code, due to the incorrect decryption, results in a valid landing pad instruction. Precisely, the probability that uniformly distributed data exactly matches a specific byte value is $\frac{1}{256}$. The `endbr64` instruction uses a 4 B instruction encoding. Thus, the probability of a decryption with a wrong key resulting in an `endbr64` instruction can be given as $(\frac{1}{256})^4$, *i.e.*, 2^{-32}.

Authenticated Encryption. Moreover, our design can also take advantage of Intel TME-MK with integrity. Intel TDX [17, 28, 30] adds support for cryptographic integrity through authenticated encryption. Here, TME-MK provides an encryption mode that associates cache lines with a cryptographic MAC. This cryptographic integrity provides detection when functions are accessed with the incorrect key. Precisely, Intel TME-MK leverages a 28-bit MAC, resulting in a probability of $1-2^{-28}$ for detecting the violation and throwing an exception [65].

7 Performance Evaluation

In this section, we provide the performance evaluation of our design. We evaluate and discuss the overhead of our design with the SPEC CPU2017 [10] benchmark suite compiled with our LLVM extension and `-O3` optimization level.

Evaluation Setup. We perform our evaluation on an Intel Xeon Gold 6530 processor with support for the Intel TME-MK memory encryption. The CPU features 32 cores, where each core has a 32 kB L1I/48 kB L1D cache and a 2 MB L2 cache. All cores share a 160 MB L3 last-level cache (LLC). Moreover, our system configuration uses 512 GB DDR5-4800 DRAM with ECC memory. The given CPU provides 6-bit keyIDs that are usable for our code encryption scheme.

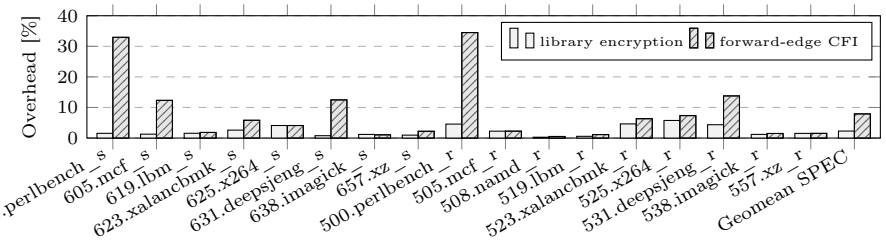

Fig. 4: The relative performance overhead of our design for SPEC CPU2017.

SPEC CPU2017 Results. For our evaluation, we benchmark our two security-hardened configurations and compare them to a baseline configuration, showcasing the performance overheads. Note that we use the *ref* input to evaluate all SPEC CPU2017 benchmarks. Our security-hardened configurations demonstrate the runtime overhead for forward-edge CFI and library code encryption, as detailed in Section 4. As our toolchain targets the hardening of C and C++ software, all Fortran benchmarks of SPEC CPU2017 are excluded. Furthermore, we exclude benchmarks with compatibility issues, e.g., the `nab` benchmark uses different function signatures for the forward declaration of external functions.

Figure 4 showcases the relative performance overhead of our design for the SPEC CPU2017 benchmark suite. We find that library code encryption imposes a low geomean overhead of 2.2 % for confining forward-edge control-flow transfers targeting security-critical library functions. Furthermore, signature-based CFI imposes a geomean overhead of 7.8 % for fine-grained control-flow enforcement. The results vary across benchmarks. We find that the overhead mainly stems from two sources: the compiler instrumentation and the increased translation lookaside buffer (TLB) pressure caused by page aliasing. For example, our code encryption scheme imposes the largest performance overhead for the `perlbench` benchmark, which performs a higher relative number of function calls and returns than other benchmarks. In addition, we use the `perf` tool to further analyze the underlying causes of the incurred overhead. The results indicate that the overhead of the library encryption reflects the overhead of our compiler instrumentation. Moreover, page aliasing reduces TLB efficiency and increases the TLB miss rate, resulting in page table walks that increase memory latency. Note that the additional overhead of the signature-based CFI (compared to library encryption) strongly correlates with the increase in TLB pressure for all benchmarks. For instance, we find that the majority of the overhead for `perlbench` is due to the increase in the number of TLB misses by an order of magnitude.

8 Discussion

In this section, we compare our design with related work on control-flow enforcement, and discuss limitations and potential future work.

8.1 Related Work

FineIBT [23] provides fine-grained forward-edge CFI using Intel Indirect Branch Tracking (IBT) with compiler support for logical integrity checks to restrict valid indirect control-flow targets. Microsoft's Control Flow Guard (CFG) [8] enables forward-edge CFI through compiler instrumentation for runtime checks to validate the destinations of indirect control-flow transfers. In contrast, our design repurposes Intel TME-MK's hardware-backed encryption (*i.e.*, correctly decrypted code) instead of instrumenting logical integrity checks in software.

Code-pointer integrity (CPI) [36] ensures forward-edge CFI by enforcing integrity for code pointers. CPI identifies sensitive pointers (*i.e.*, code pointers and pointers that may access code pointers indirectly) through static analysis and instruments the program to store sensitive pointers and associated metadata at a protected memory region. The metadata of sensitive pointers is then checked on pointer dereferences. CPI also provides a safe stack [36] for proven-safe objects.

In addition to CFI measures through logical integrity, CFI techniques based on cryptographic primitives have also been explored in prior work. For instance, ARM pointer authentication (PAuth) [51] and CCFI [44] provide CFI measures through the use of cryptographic message authentication codes (MACs). Also, PointGuard [19] enables pointer protection through the encryption of pointers.

The overall concept of ARM PAuth is to protect (code and data) pointers stored in memory from corruption [39, 40]. Therefore, the cryptographic MAC, so-called pointer authentication code (PAC), of the pointer is generated and encoded into the upper bits of the pointer. This PAC ensures the pointer's integrity while stored in memory. Moreover, after loading the pointer from memory, the pointer is authenticated, detecting any potential manipulation (with probabilistic security depending on the size of the MAC [51]). ARM PAuth has been extensively studied, resulting in the outlining of potential weaknesses [13], e.g., PAC reuse [32, 39] or PAC forgery [6]. Additionally, the PACMAN [52] vulnerability showcased how to brute-force PAC values through speculative execution.

Intel TME-MK has been used to help protect data in memory, e.g., by enforcing memory safety [57, 58] and in-process isolation [65]. In addition, EC-CFI [48] presents control-flow integrity counteracting fault attacks by combining Intel TME-MK with the Intel virtualization technology. However, it is important to clarify that this approach is designed to protect against fault attacks [7]. This threat model includes a physical attacker that actively induces faults into the processor, e.g., through laser fault injection [7]. Contrarily, our approach targets a software attacker that exploits memory safety errors to hijack the control flow.

Other CFI schemes introduce custom hardware extensions for code encryption to primarily counteract fault attacks [18, 24, 47, 68]. For example, SCFP [68] offers instruction granular control-flow protection by integrating an additional pipeline stage into the processor to decrypt the instructions during runtime. This protects the control flow against logical and physical attacks since the tampering of instructions leads to incorrect decryption and execution of pseudo-random instructions. In contrast, our design targets software attackers, while SCFP's instruction granular protection primarily focuses on counteracting fault attacks.

8.2 Limitations and Future Work

Our code encryption design requires an increased number of TLB entries, as each encryption key used within a page requires a separate TLB entry. This increases the TLB pressure, leading to a decrease in performance. Future optimizations can use 2 MB-sized pages to lessen the overhead incurred by TLB pressure. Moreover, our design does not address fault injection attacks. An adversary with fault injection capabilities or physical access to the CPU poses a potential threat; thus, orthogonal countermeasures might be required.

This work implements signature-based CFI as a proof-of-concept; however, our generic design allows to enforce different security policies on top of the underlying code encryption mechanism. Future work could explore other CFI policies [42, 64, 67] to limit control-flow transfers. In addition, the AMD Secure Memory Encryption (SME) [33] feature enables memory encryption on AMD machines, leveraging a single encryption key. Recently, AMD also introduced Secure Memory Encryption Multi-Key (SME-MK) [3, 4], an extension of AMD SME that supports multiple encryption keys. Future work could explore the implementation of a comparable code encryption scheme on AMD CPUs.

9 Conclusion

In this paper, we presented code encryption with Intel TME-MK, a novel approach for fine-grained control-flow enforcement on off-the-shelf x86 machines. We repurpose the Intel TME-MK hardware feature to encrypt individual functions and to associate control-flow operations with designated encryption keys. This restricts control-flow transfers solely to destinations that are permitted by our security policies, *i.e.*, encrypted with their respective encryption key.

This way, we enforce software hardening techniques for forward-edge CFI and library encryption by securing executable code via TME-MK's encryption. More concretely, our generic scheme allows us to efficiently encrypt individual functions through the use of up to 32K encryption keys. Control-flow hijacking leads to incorrect decryption and, thus, to garbled code, preventing software attacks that aim to illegitimately divert the program's execution flow through function pointer manipulation. This cryptographic restriction of control-flow transfers also achieves detection through wrongly decrypted IBT landing pads.

We implement a prototype of our security-hardened toolchain, consisting of an LLVM compiler extension, a modified ELF loader, and a kernel patch. Our performance evaluation showcases a geomean overhead of 7.8 % for forward-edge CFI and 2.2 % for library encryption using the SPEC CPU2017 benchmark suite.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback that improved this work. This project has received funding from the Austrian Research Promotion Agency (FFG) via the AWARE project (FFG grant number 891092) and the RESIST project (FFG grant number 915106). Additional funding was provided by a generous gift from Intel.

References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-Flow Integrity. In: CCS (2005)
2. Aktas, E., Cohen, C., Eads, J., Forshaw, J., Wilhelm, F.: Intel Trust Domain Extensions (TDX) Security Review. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf (2023), Accessed: 2024-06-10
3. AMD: 4th Gen AMD EPYC Processor Architecture. https://www.amd.com/en/products/processors/server/epyc/4th-generation-architecture.html (2023), Accessed: 2024-05-27
4. AMD: AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf (2025), Accessed: 2025-02-26
5. Arm: Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487 (2025), Accessed: 2025-02-26
6. Azad, B., Google Project Zero: Examining Pointer Authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html (2019), Accessed: 2024-06-10
7. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE **94**, 370–382 (2006)
8. Biondo, A., Conti, M., Lain, D.: Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets. In: NDSS (2018)
9. Bletsch, T.K., Jiang, X., Freeh, V.W., Liang, Z.: Jump-Oriented Programming: A New Class of Code-Reuse Attack. In: ASIACCS (2011)
10. Bucek, J., Lange, K., von Kistowski, J.: SPEC CPU2017: Next-Generation Compute Benchmark. In: ICPE (2018)
11. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In: CCS (2008)
12. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-Flow Integrity: Precision, Security, and Performance. ACM Computing Surveys **50**, 16:1–16:33 (2017)
13. Cai, Z., Zhu, J., Shen, W., Yang, Y., Chang, R., Wang, Y., Li, J., Ren, K.: Demystifying Pointer Authentication on Apple M1. In: USENIX Security (2023)
14. Carlini, N., Barresi, A., Payer, M., Wagner, D.A., Gross, T.R.: Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In: USENIX Security (2015)
15. Carlini, N., Wagner, D.A.: ROP is Still Dangerous: Breaking Modern Defenses. In: USENIX Security (2014)
16. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., Winandy, M.: Return-Oriented Programming without Returns. In: CCS (2010)
17. Cheng, P., Ozga, W., Valdez, E., Ahmed, S., Gu, Z., Jamjoom, H., Franke, H., Bottomley, J.: Intel TDX Demystified: A Top-Down Approach. ACM Computing Surveys **56**, 238:1–238:33 (2024)
18. de Clercq, R., de Keulenaer, R., Coppens, B., Yang, B., Maene, P., de Bosschere, K., Preneel, B., de Sutter, B., Verbauwhede, I.: SOFIA: Software and Control Flow Integrity Architecture. In: DATE (2016)
19. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In: USENIX Security (2003)
20. Daemen, J., Rijmen, V.: The Block Cipher Rijndael. In: CARDIS (1998)

21. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography (2002)
22. Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., Halderman, J.A.: The Matter of Heartbleed. In: IMC (2014)
23. Gaidis, A.J., Moreira, J., Sun, K., Milburn, A., Atlidakis, V., Kemerlis, V.P.: FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In: RAID (2023)
24. Gousselot, T., Dutertre, J., Potin, O., Rigaud, J.: Code Encryption for Confidentiality and Execution Integrity down to Control Signals. In: HOST (2025)
25. Graham-Cumming, J.: Incident report on memory leak caused by Cloudflare parser bug. https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug (2017), Accessed: 2024-06-10
26. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest We Remember: Cold Boot Attacks on Encryption Keys. In: USENIX Security (2008)
27. Intel: Intel Architecture Memory Encryption Technologies. https://www.intel.com/content/www/us/en/content-details/679154/intel-architecture-memory-encryption-technologies-specification.html (2022), Revision 1.4, Accessed: 2023-01-31
28. Intel: Intel Trust Domain Extensions. https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf (2022), Accessed: 2024-05-27
29. Intel: Runtime Encryption of Memory with Intel Total Memory Encryption-Multi-Key (Intel TME-MK). https://www.intel.com/content/www/us/en/developer/articles/news/runtime-encryption-of-memory-with-intel-tme-mk.html (2022), Accessed: 2024-05-27,
30. Intel: Architecture Specification: Intel Trust Domain Extensions (Intel TDX) Module. https://cdrdv2-public.intel.com/733568/tdx-module-1.0-public-spec-344425005.pdf (2023), Accessed: 2024-05-27
31. Intel Labs: TME-MK-i for Memory Safety. https://github.com/intellabs/tme-mk-fine-grained-encryption-integrity (2024), Accessed: 2024-05-20
32. Ismail, M., Quach, A., Jelesnianski, C., Jang, Y., Min, C.: Tightly Seal Your Sensitive Pointers with PACTight. In: USENIX Security (2022)
33. Kaplan, D., Powell, J., Woller, T.: AMD Memory Encryption. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf (2021), Accessed: 2024-05-27
34. Kim, Y., Daly, R., Kim, J.S., Fallin, C., Lee, J., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In: ISCA (2014)
35. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution. In: S&P (2019)
36. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-Pointer Integrity. In: OSDI (2014)
37. Lattner, C., Adve, V.S.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO (2004)
38. LeMay, M., Rakshit, J., Deutsch, S., Durham, D.M., Ghosh, S., Nori, A., Gaur, J., Weiler, A., Sultana, S., Grewal, K., Subramoney, S.: Cryptographic Capability Computing. In: MICRO (2021)

39. Liljestrand, H., Nyman, T., Gunn, L.J., Ekberg, J., Asokan, N.: PACStack: an Authenticated Call Stack. In: USENIX Security (2021)
40. Liljestrand, H., Nyman, T., Wang, K., Perez, C.C., Ekberg, J., Asokan, N.: PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In: USENIX Security (2019)
41. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: USENIX Security (2018)
42. Lu, K., Hu, H.: Where Does It Go?: Refining Indirect-Call Targets with Multi-Layer Type Analysis. In: CCS (2019)
43. Martin, L.: XTS: A Mode of AES for Encrypting Hard Disks. IEEE Security & Privacy **8**, 68–69 (2010)
44. Mashtizadeh, A.J., Bittau, A., Boneh, D., Mazières, D.: CCFI: Cryptographically Enforced Control Flow Integrity. In: CCS (2015)
45. Miller, M.: Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. `https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf` (2019), Accessed: 2023-02-26
46. Murdock, K., Oswald, D.F., Garcia, F.D., Bulck, J.V., Gruss, D., Piessens, F.: Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: S&P (2020)
47. Nasahl, P., Mangard, S.: SCRAMBLE-CFI: Mitigating Fault-Induced Control-Flow Attacks on OpenTitan. In: GLSVLSI (2023)
48. Nasahl, P., Sultana, S., Liljestrand, H., Grewal, K., LeMay, M., Durham, D.M., Schrammel, D., Mangard, S.: EC-CFI: Control-Flow Integrity via Code Encryption Counteracting Fault Attacks. In: HOST (2023)
49. National Security Agency: NSA Cybersecurity Information Sheet: Software Memory Safety. `https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF` (2022), Accessed: 2023-02-26
50. Prince, M.: Quantifying the Impact of "Cloudbleed". `https://blog.cloudflare.com/quantifying-the-impact-of-cloudbleed` (2017), Accessed: 2024-06-10
51. Qualcomm: Pointer Authentication on ARMv8.3. `https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf` (2017), Accessed: 2023-02-26
52. Ravichandran, J., Na, W.T., Lang, J., Yan, M.: PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In: ISCA (2022)
53. Rebert, A., Kern, C.: Secure by Design: Google's Perspective on Memory Safety. Tech. rep., Google Security Engineering (2024)
54. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-Oriented Programming: Systems, Languages, and Applications. ACM Transactions on Privacy and Security **15**, 2:1–2:34 (2012)
55. Rogaway, P., Bellare, M., Black, J., Krovetz, T.: OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In: CCS (2001)
56. Röttger, S.: Control-flow Integrity in V8. `https://v8.dev/blog/control-flow-integrity` (2023), Accessed: 2024-06-10
57. Schrammel, D., Sultana, S., Grewal, K., LeMay, M., Durham, D.M., Unterguggenberger, M., Nasahl, P., Mangard, S.: MEMES: Memory Encryption-Based Memory Safety on Commodity Hardware. In: SECRYPT (2023)

58. Schrammel, D., Unterguggenberger, M., Lamster, L., Sultana, S., Grewal, K., LeMay, M., Durham, D.M., Mangard, S.: Memory Tagging using Cryptographic Integrity on Commodity x86 CPUs. In: EuroS&P (2024)
59. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In: CCS (2007)
60. Shanbhogue, V., Gupta, D., Sahita, R.: Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In: HASP (2019)
61. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: Eternal War in Memory. In: S&P (2013)
62. Tang, A., Sethumadhavan, S., Stolfo, S.J.: CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: USENIX Security (2017)
63. Taylor, A., Whalley, A., Jansens, D., Oskov, N.: An update on Memory Safety in Chrome. `https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html` (2021), Accessed: 2023-02-26
64. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In: USENIX Security (2014)
65. Unterguggenberger, M., Lamster, L., Schrammel, D., Schwarzl, M., Mangard, S.: TME-Box: Scalable In-Process Isolation through Intel TME-MK Memory Encryption. In: NDSS (2025)
66. Unterguggenberger, M., Schrammel, D., Lamster, L., Nasahl, P., Mangard, S.: Cryptographically Enforced Memory Safety. In: CCS (2023)
67. van der Veen, V., Göktas, E., Contag, M., Pawlowski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., Giuffrida, C.: A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In: S&P (2016)
68. Werner, M., Unterluggauer, T., Schaffenrath, D., Mangard, S.: Sponge-Based Control-Flow Protection for IoT Devices. In: EuroS&P (2018)