iVault: Architectural Code Concealing Techniques to Protect Cryptographic Keys

George Christou[1], Giorgos Vasiliadis[2,3], Apostolis Zarras[3,4], and Sotiris Ioannidis[1,3]

[1] Technical University of Crete, Chania, Greece
[2] Department of Management Science and Technology, Hellenic Mediterranean University, Agios Nikolaos, Greece
[3] Foundation for Research and Technology – Hellas, Heraklion, Greece
[4] University of Piraeus, Piraeus, Greece

Abstract. Memory corruption bugs remain a significant concern in applications developed using memory-unsafe languages, such as C/C++. Adversaries can exploit these bugs and perform arbitrary read and write operations. These arbitrary reads can target cryptographic keys, severely compromising their secure operation. In this paper, we introduce `iVault` a lightweight approach to securely store private and secret cryptographic keys. `iVault` encodes cryptographic keys within the machine instruction immediates and leverage architectural mechanisms to protect the `.text` segment from being disclosed. We assess `iVault` in terms of performance and code size expansion, and we show that it represents a viable solution for safeguarding cryptographic keys.

Keywords: Hardware Assisted Security · Key Management

1 Introduction

Modern cryptography is typically used to protect sensitive or private data from being intercepted and/or leaked when transferred over the network or stored in memory. The adoption of cryptographic hardware extensions within commodity CPUs has reduced the overheads imposed from cryptographic operations on data, making cryptography mainstream and widely used in diverse set of applications [16, 22]. Besides performance, cryptography is only effective in terms of security, as long as the cryptographic keys are adequately protected [15]. In fact, the process of securely storing and managing cryptographic keys used for encryption and decryption, known as key management, forms the basis for the end-to-end security of a system.

Key management is often considered quite challenging though, especially in cases where the keys must be accessed frequently, such as real-time processing or streaming applications. This is further exacerbated by the fact that the majority of crypto libraries are written in low-level languages, such as C/C++, mainly to access the architectural extensions available in modern CPUs for accelerating

cryptography. Unfortunately, these low-level languages are not memory safe, and thus, memory corruption bugs can allow attackers to manipulate the memory contents and access sensitive information, such as secret or private cryptographic keys. One such example is the Heartbleed bug that allowed buffer over-reads due to improper input validation [14]. This bug can be exploited to disclose the cryptographic keys from the main memory and has affected an enormous amount of computing systems and users, due to the popularity of the affected library [8].

To tackle this problem, several academic works have been proposed to define safe regions for storing sensitive data [23, 24]. These works mostly focus on intra-process isolation, since traditional operating systems offer strong isolation guarantees between processes. Among these, the deployment of memory safety mechanisms to the C and C++ programming languages can restrict memory accesses and eliminate memory corruption vulnerabilities. However, these mechanisms lack practicality due to high performance overhead. Other approaches utilize hardware-based Trusted Execution Environments (TEEs), such as Intel SGX [17], AMD SEV [1], or ARM TrustZone [2]. These TEEs provide secure and isolated environments where the code and the keys can reside, preventing unauthorized access or tampering from any outside entity. Still, current hardware-based TEEs do not guarantee the memory safety of code that has been developed with memory unsafe languages such as C and C++. As a consequence, programs running inside TEEs face the same memory corruption vulnerabilities as traditional software, allowing attackers to violate their confidentiality.

In this work, we propose `iVault`, a lightweight approach to securely store private and secret cryptographic keys as immediates in machine code. The machine code is then protected using hardware-assisted techniques that are able to *hide* the `.text` segment of binaries. By doing so, the encoded keys are sufficiently protected in case of memory corruption vulnerabilities that allow an attacker to disclose arbitrary memory regions. `iVault` implements a representative set of popular cryptographic algorithms, in which the cryptographic keys are encoded as immediates in the corresponding machine code. `iVault` also provides a set of API functions that allow to easily encode the cryptographic keys on demand.

We implement `iVault` in two different architectures: SPARC and x86. For SPARC, we employ a modified Leon3 processor that is equipped with Instruction Set Randomization. In the case of the x86, we take advantage of Intel Memory Protection Keys (MPKs), a feature available on recent Intel processor models. Our evaluation shows that `iVault` can effectively prevent the exploitation of memory disclosure bugs that target cryptographic keys, with an average overhead of 0.36% in x86 and 8.93% in SPARC.

2 Background

In this section we briefly describe the two architectural features that we utilize for the implementation of `iVault` in regards to the protection of the .text segment: Instruction Set Randomization and Memory Protection Keys.

2.1 Instruction Set Randomization

Instruction Set Randomization (ISR) aims to fortify applications against code injection attacks [20]. Its fundamental concept involves the randomization of the instruction set of each process through encryption. Thus, any injected code will likely fail to execute, as decryption takes place before entering into the processor's instruction cache, potentially resulting in invalid code. Originally, ISR was not conceived to defend against Code Reuse Attacks (CRAs) such as Return Oriented Programming (ROP) [25] and Jump Oriented Programming (JOP) [6]. However, ISR schemes with strong encryption can thwart CRAs by obstructing the disclosure of useful gadgets. In ISR, the `.text` segment is encrypted, causing data accesses on this section to return encrypted bytes. In `iVault`, we leverage ASIST [10], a modified Leon3 processor [13], which provides three distinct encryption modes for ISR: XOR, Transposition, and AES. In `iVault` we opt for the AES variant due to its ability to also defend against CRAs and its resilience against cryptographic attacks such as known-plaintext-ciphertext.

Overall, `iVault` relies on the capability of ISR to prevent the disclosure of the `.text` segment, thereby enabling us to *conceal* cryptographic keys. Consequently, any arbitrary memory read attempt will fail to unveil this information to potential attackers.

2.2 Memory Protection Keys

Memory Protection Keys (MPK) is a mechanism introduced recently in Intel processors [18]. It allows user-space applications to directly modify the access rights on groups of memory pages. Each group of pages is associated with a unique key, with applications capable of managing up to 16 page groups. The access rights for each page group are delineated in a thread-local and user-accessible register known as the Protection Keys Rights Register (for users), denoted as `%pkru`. Given that the `%pkru` register is specific to each thread, MPK facilitates a per-thread perspective of the process's memory. For instance, distinct application threads possess different access rights configured for each key within their respective `%pkru` register.

If a memory page is marked as executable in the page table but configured with `no access` in the `%pkru` register, the memory page is treated as execute-only. This occurs since any attempt at data access will lead to a mismatch between the rights specified in the page table and those in the `%pkru` register. Linux leverages MPK to support execute-only memory pages. Invoking `mprotect` with only `PROT_EXEC` specified as permissions will result in the allocation of a protection key associated with the memory pages passed to `mprotect`. Subsequently, the `%pkru` register will be set to `DISABLE_ACCESS` for the newly allocated protection key, while the page table rights will be configured as executable and readable. Any attempt to access execute-only pages, except for instruction fetching, will trigger a memory violation exception.

In `iVault`, we leverage execute-only memory to achieve similar properties as with ISR. By preventing read access on the `.text` segment, the embedded keys will not be disclosed due to memory corruption bugs.

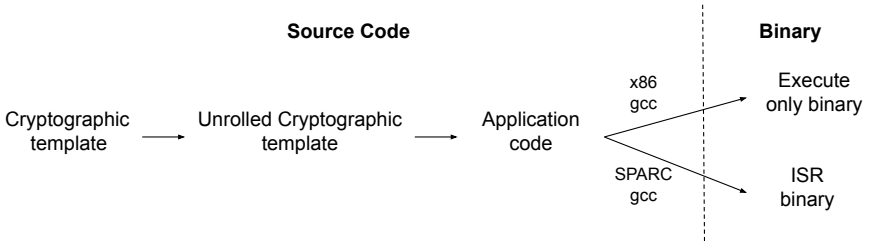


Fig. 1: `iVault` template inclusion in applications.

3 Threat model

Our threat model assumes adversaries can exploit memory and type unsafety in applications that utilize cryptography. Consequently, an attacker can exploit memory corruption vulnerabilities to enable arbitrary memory read capabilities. These vulnerabilities can lead to unauthorized access to sensitive data. Finally, we consider side-channel attacks and hardware faults as out of scope.

To secure keys stored in the `.text` segment, specific OS and hardware features are necessary. We presume the W^X policy is enforced, and the application excludes self-modifying code. Our cryptographic implementations are designed to coexist with other security mechanisms, such as AppArmor and RELRO, which can further enhance application protection. The hardware must include Memory Protection Keys or a functionally equivalent mechanism. For our Leon3 implementation, we employ ISR. Although our required hardware feature is not as standard as our OS prerequisites, it is included in the latest Intel server CPU series. Additionally, the MPK functionality can be emulated through memory tagging, which is available in ARM's latest processor series [4].

4 Design

We outline the design of `iVault` and the cryptographic suite that implements. For each cryptographic algorithm, `iVault` uses templates that can be employed to encode the keys as immediate operands at various stages of the application's life-cycle, such as post-compilation, program initialization, or program execution.

4.1 Source code transformations

To correctly embed the keys for each cryptographic algorithm, it is essential to ensure that every cryptographic operation is implemented as a block of instructions with only one entry and exit point, namely *basic block*. This is required because, otherwise, it would be impractical to modify the instruction immediates at runtime, on every iteration of an encryption algorithm (e.g., AES requires 10 loops). To unroll the loops, we serialize and transform each round of the block ciphers implemented in `iVault` as shown in Figure 1, ensuring that all data operations are executed without any branch present.

4.2 Cryptographic key set

The cryptographic keys are set at various stages of an application's life cycle. Initially, the immediates of the key instructions are set to dummy values. Each cryptographic function is analyzed to produce a map of the key byte locations. The real key is embedded by iterating through the map that locates key part positions within the binary. At this stage, instructions that carry key parts are patched to incorporate the key used during the application's execution. Next, the key can be modified during the application's runtime, a functionality that varies with the architecture in which `iVault` is implemented. For the x86 architecture, the function included in the cryptographic library modifies the key parts at runtime, utilizing the key map. In contrast, in our ISR-based implementation, where the `.text` area is encrypted, and the ISR key is stored in the kernel process table; we depend on the ISR-aware operating system to correctly encrypt the rewritten cryptographic functions—those patched with the new encryption key. As we show in Section 6, encoding the cryptographic keys in the instructions immediates reduces the memory accesses required during each cryptographic operation.

5 Implementation

We now describe the implementation details of `iVault` for symmetric and asymmetric key cryptography, respectively.

5.1 Symmetric cryptography

We implement two block ciphers to evaluate the design of `iVault`: XTEA and AES. XTEA is a block cipher encryption algorithm that operates on 64-bit data blocks using a 128-bit key and employs a Feistel network structure [28]. This involves splitting the input data into two halves, performing a series of operations on each, and then merging the outcomes. The key generates multiple round keys that alter the data during each Feistel cycle, with a typical configuration of 64 rounds, enhancing its security. Due to its simplicity, speed, and compact code size, XTEA is frequently chosen for embedded systems. We unroll the algorithm loops for the cryptographic key embedding and replace key array accesses with dummy constant values. We then compile this modified code into a binary that embeds these dummy parts as immediates in the machine instructions. Now, a script associates each key part with the corresponding instruction and patches the binary with the real key.

We also implement AES algorithm in ECB mode, in which the input data is partitioned into fixed-size blocks (128 bits for AES), that are encrypted separately using the same key. This direct encryption process requires no extra data or computations beyond the key and the input data. Identical plaintext blocks will consistently yield the same ciphertext blocks. Our approach to implement ECB is similar to XTEA: we unroll all rounds, use constant values as a dummy key, and later patch the actual key to be used in the targeted binary.

5.2 Asymmetric cryptography

In `iVault`, we incorporate the 256-bit Elliptic Curve Digital Signature Algorithm (ECDSA) [19], which is a digital signature algorithm that leverages elliptic curve cryptography (ECC). ECDSA is widely used to verify the authenticity, integrity, and non-repudiation of digital messages and transactions. The ECDSA process involves a pair of public and private keys; the private key generates digital signatures, whereas the public key is used for their verification.

The algorithm initiates by creating a random number, known as a nonce, to determine a point on an elliptic curve. This point is then used to calculate the signature, comprising a pair of integers: the x-coordinate and a value derived from the nonce and the elliptic curve point. For signature verification, the receiver employs the sender's public key to compute the corresponding elliptic curve point and verifies its congruence with the point involved in the signature creation. The receiver then validates the signature using the same mathematical procedures originally used to generate it. In `iVault`, we have implemented the sign function, the sole function that employs the private key. A notable distinction from the block ciphers we have developed is that ECDSA utilizes a large integer for the key rather than a byte array.

5.3 Architectural specific implementations

We implement `iVault` on two different architectures: x86 and SPARC. A notable distinction in our approach is that our x86 processor operates on a 64-bit architecture, whereas our SPARC processor is a 32-bit. This architectural difference does not affect our block ciphers; however, for ECDSA, the key division involves four instructions on the x86 architecture and eight on the SPARC architecture.

We also employ different architectural extensions specific to each architecture to enhance security. For the x86 implementation, we use MPK to designate the page containing the cryptographic functions as execute-only. Conversely, we use ISR for the SPARC implementation to encrypt the entire `.text` segment of the application that employs the `iVault` cryptographic suite. A notable difference is that in our ISR implementation, the whole `.text` segment is protected, whereas, in x86, only the pages containing the cryptographic operations are configured as execute only. Thus, in our x86 implementation, we have to page-align the first sensitive function and the first non-sensitive function to avoid the co-existence of cryptographic functions with normal ones on the same page.

5.4 Changing encryption keys

In each of our implementations, we have developed functions that enable the cryptographic key to be changed at runtime. For the x86 architecture, this function employs *mprotect* to mark the page containing the cryptographic function as writable. Then, the new key is patched into the immediates of the instructions using the map we produced during the initial stages of the implementation. This

map includes the address offset of key parts in the cryptographic functions, calculating the true address by adding the function's base load address to the offset in the map. After patching the new key, the key change function uses *mprotect* again to set the access rights to execute only.

In our ISR implementation, direct modification of the cryptographic functions' code is not feasible, as it is encrypted, and the key resides solely in kernel space. Therefore, we first use *mprotect* to make the pages containing the cryptographic functions writable and then completely overwrite them with a new function embedded with the new cryptographic key. We then use the *asist_ encrypt* system call to encrypt the new function with the process's ISR key. Finally, we call *mprotect* once more to set the access rights to executable.

6 Evaluation

We evaluate `iVault` in terms of security and in terms of performance. To achieve this, we use a set of popular cryptographic algorithms both in standalone setups and real-world scenarios. Our aim is to answer the following evaluation questions:

EQ1: Can `iVault` protect cryptographic keys against memory disclosure?

EQ2: What is the impact of `iVault` on the binary code size?

EQ3: What is the impact of `iVault` on the runtime performance?

6.1 Security analysis (EQ1)

We assess the resilience of `iVault` by delineating attacks conforming to our threat model, described in Section 3. We also implement synthetic attack scenarios to showcase how `iVault` prevents the extraction of cryptographic keys from an application.

Arbitrary memory reads. In `iVault`, the cryptographic keys reside only within the `.text` area of an application, which is sufficiently isolated in both implementations (i.e., x86 and SPARC). On x86 architectures, accesses to the `.text` segment are prohibited because it is configured with execute-only permissions. Similarly, in our customized SPARC V8 processor, the `.text` segment is encrypted. Consequently, any attempts at arbitrary read primitives, such as buffer over-reads and use-after-free, will fail to disclose the cryptographic keys.

Extracting intermediate states. In `iVault`, each cryptographic algorithm is realized as a basic block, implemented as a code block without any branches. During every cryptographic operation, the function executes independently of any other variables in the application. At the epilogue of each function, we ensure that any intermediate states generated during the cryptographic operation are nullified and, therefore, cannot be retrieved (e.g., through arbitrary reads accessible from other code blocks).

Code reuse attacks. A major class of exploitation techniques that target memory vulnerabilities is code reuse attacks. In this scenario, the attacker exploits

Table 1: The code size growth (in bytes) of `iVault` in the execute-only memory version (x86).

Bench	Vanilla	iVault	Overhead
`XTEA`	1205	16981	1309.21%
`AES`	6789	13189	94.27 %
`ECC`	30581	34677	13.39 %
`SQLite`	868564	881412	1.48 %

Table 2: The code size growth (in bytes) of `iVault` in the Instruction Set Randomization version (SPARC).

Bench	Vanilla	iVault	Overhead
`XTEA`	29496	44428	50.62%
`AES`	49852	56624	13.58 %
`ECC`	87780	94312	7.44 %
`SQLite`	1207720	1221992	1.18 %

arbitrary memory writes to overwrite control-flow variables. Still, an attacker might use the `iVault`'s API to alter the cryptographic keys. Nonetheless, this action would not disclose any data encrypted with previous keys. For code reuse attacks to succeed, the attacker must know in advance the location of the code snippets necessary to implement the payload. However, the `.text` segments of `iVault` are unreadable, rendering the attacker unaware of the location of useful code. The idea of removing read access from the code segment has been studied [3]. Advanced code reuse attack techniques have demonstrated that applications can still be exploited even without knowledge of the layout of the `.text` segment [5]. However, concealing the code segment significantly raises the bar for a successful code reuse attack.

Take Away: `iVault` can effectively protect cryptographic keys in applications against arbitrary reads.

6.2 Overhead on the code size (EQ2)

As we mention in Section 4.1, `iVault` unrolls each and every loop contained in the targeted cryptographic algorithm in order to have only a *large* basic block that contains all the instructions that will encrypt or decrypt a block of data. This inevitably expands the `.text` segment size of the final binary, which we measure by comparing the original versions of the algorithms utilized for prototyping `iVault` with the modified binaries. As depicted in Table 1, the growth of the `.text` segment in our execute-only memory implementation surpasses that of our ISR implementation. This disparity stems from MPKs operating on a per-page granularity. Thus, we page-aligned the first *sensitive* function and the first subsequent non-sensitive function of the binary. In the case of `XTEA` algorithm, this increases `.text` by half a page (approximately). One could argue that the execute-only memory could be applied to the whole `.text` segment in order to reduce growth. However, this is not always possible since compilers may emit data (e.g., jump tables) in the `.text` segment.

We note that the ISR implementation does not require page-aligning sensitive functions, since the whole application is protected. To protect binaries, we compile them statically and without jump tables and strict aliasing (emit data

to the `.text` segment). We also use the same compilation flags for both vanilla and `iVault` binaries in order to gather consistent measurements for code size growth and runtime performance. As we observe in Table 2, the percentage of the code added is lower since the binaries are statically linked and are amortized in both cases when the implementations are included in larger code bases (i.e. in ECC and SQLite).

Take Away: `iVault` code size overhead is not prohibitive in applications with large code base.

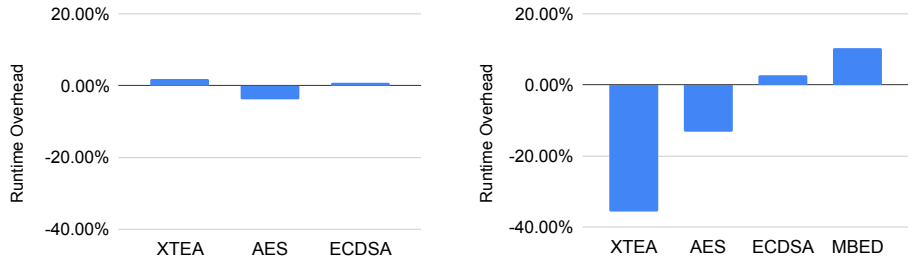
6.3 Runtime performance evaluation (EQ3)

To assess the runtime performance of `iVault`, we run benchmarks on popular cryptographic algorithms and real application scenarios. The standalone execution of cryptographic operations can pinpoint the performance impact of our modifications, while large applications provide us more insights with regards to the real-world expected impact.

Setup. We evaluate `iVault` on x86 and SPARC V8 architectures. Our x86 system features an Intel Core i9-10900 CPU with 32GB RAM and runs Linux version 5.4.0-84. To ensure consistent measurements, we disable frequency scaling and turbo boost. For evaluating `iVault` on the ISR-enabled SPARC V8 processor, we utilize a Leon3 processor configured with a 64-KB (4-way associative) instruction cache. The cache operates with a single level and follows a pure Harvard architecture. We synthesize and map the ISR-enabled Leon3 processor onto a Xilinx XUPV5 ML509 FPGA board with 256 MB DDR2 SDRAM memory, operated at 80-MHz core clock frequency. Our measurements include the overhead imposed by the ISR, as we run the vanilla results on a non-ISR Leon3 with identical configuration (frequency, RAM).

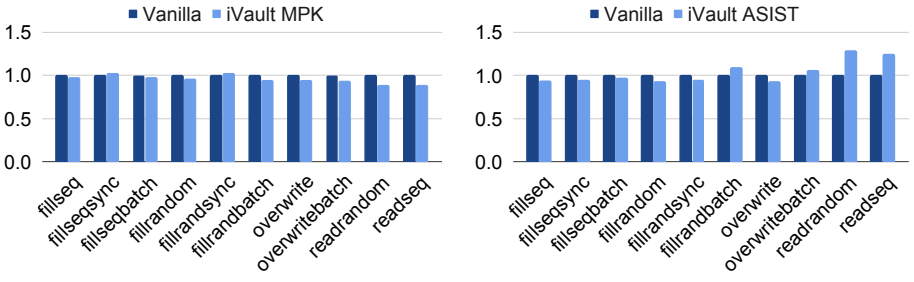
Algorithms and applications. We first evaluate `iVault` by executing standalone cryptographic functions of individual algorithms. We measure the execution time of `iVault` by executing 1M cryptographic operations (encryption and decryption) for XTEA and AES and 1K ECDSA operations. We then modify `SQLite` and Mbed TLS to validate our approach in real-world scenarios further. Our modified SQLite utilizes AES to provide fully encrypted databases, while in Mbed TLS, we alter the ECDSA signing and verification functions. For SQLite, we run the default benchmark values in x86 (100K entries), while in Leon3, we configured the benchmark to operate on 100 entries due to the system's limited capabilities.

Our findings suggest that `iVault` is a practical security strategy. In x86 (Figure 2a), the runtime performance ranges between -3.8% to 2% and is affected negatively by the code size growth and reduced cache locality (due to page-alignment) and positively by the reduced number of memory operations required for each cryptographic operations. This is especially prevalent in our SPARC implementation (Figure 2b), where the reduced amount of memory operations



(a) x86 (b) SPARC

Fig. 2: The runtime overheads of `iVault` for different cryptographic algorithms.



(a) Execute-only Memory (x86) `iVault` (b) ISR (SPARC) `iVault`

Fig. 3: The performance overheads of `iVault` when running SQLite benchmarks

amortize the overhead imposed due to ISR, while ECC and Mbed TLS are negatively affected since they do not stress the memory subsystem (128-bit*10 round keys for AES vs. 256-bit private key for ECDSA).

In SQLite, both implementations enhance the performance of most benchmarks(Figure 3). Slowdowns can be observed in our ISR implementation (Figure 3b) when random and read sequences are introduced. This might be because SQLite's code size is two times larger than Leon3's instruction cache, and the cache evictions will require the invocation of the ISR unit when the instructions are cached again.

Take Away: `iVault` is practical in terms of runtime performance and benefits from the reduced stress on the memory subsystem.

7 Related Work

A number of works aim to protect cryptographic keys from memory bugs. PixelVault [27] and GRIM [21] use similar code concealing techniques, but for GPU programs. Other frameworks have been designed to safeguard data. For instance,

AEGIS framework [26] supports secure operational environments and essential trusted computing functionalities for handling security-sensitive applications. Similarly, Bastion [9] integrates a memory authentication mechanism into the microprocessor, delivering safeguards against software and physical threats. These academic proposals paved the way to include TEEs in commodity processors. Intel Software Guard eXtensions (SGX) [11] aim to provide integrity and privacy guarantees to security-sensitive applications while narrowing the trusted computing base to just the processor's chip. Proposals like Sanctum [12] and MI6 [7] aim to exceed Intel SGX's security capabilities for the RISC-V CPU architecture but also add safeguards against micro-architectural side-channels.

8 Conclusions

We propose `iVault`, a novel mechanism to safeguard cryptographic keys by utilizing architectural extensions for code concealment. Our strategy was implemented on both x86 and SRARC architectures, demonstrating the advantages of execute-only memory and Instruction Set Randomization in protecting cryptographic keys. Our findings indicate that even though `iVault` leads to an increase in the size of the `.text` segment, it also decreases memory operations and enhances the performance of cryptographic functions.

Acknowledgements This work has been supported by the European Union's Horizon Europe research and innovation programme under grant agreements No. 101070599 (SecOPERA), 101120962 (RESCALE), and 101120726 (dAIEDGE). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or REA. Neither the European Union nor the granting authority can be held responsible for them.

References

1. AMD: Secure Encrypted Virtualization (SEV). `https://www.amd.com/en/developer/sev.html`
2. ARM: Trustzone, `https://www.arm.com/products/security-on-arm/trustzone`
3. Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., Pewny, J.: You can run but you can't read: Preventing disclosure exploits in executable code. In: ACM SIGSAC Conference on Computer and Communications Security (2014)
4. Bannister, S.: Memory tagging extension: Enhancing memory safety through architecture. ARM community (2018)
5. Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., Boneh, D.: Hacking blind. In: IEEE Symposium on Security and Privacy (2014)
6. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-Oriented Programming: A New Class of Code-Reuse Attack. In: ACM ASIACCS (2011)
7. Bourgeat, T., Lebedev, I., Wright, A., Zhang, S., Arvind, Devadas, S.: Mi6: Secure enclaves in a speculative out-of-order processor. In: IEEE/ACM Micro (2019)
8. Carvalho, M., DeMott, J., Ford, R., Wheeler, D.A.: Heartbleed 101. IEEE security & privacy **12**(4), 63–67 (2014)

9. Champagne, D., Lee, R.B.: Scalable architectural support for trusted software. In: International Symposium on High-Performance Computer Architecture (2010)
10. Christou, G., Vasiliadis, G., Papaefstathiou, V., Papadogiannakis, A., Ioannidis, S.: On architectural support for instruction set randomization. ACM Transactions on Architecture and Code Optimization (TACO) **17**(4), 1–26 (2020)
11. Costan, V., Devadas, S.: Intel SGX explained. Cryptology ePrint Archive (2016)
12. Costan, V., Lebedev, I., Devadas, S.: Sanctum: Minimal hardware extensions for strong software isolation. In: USENIX Security Symposium (2016)
13. Daněk, M., Kafka, L., Kohout, L., Sỳkora, J., Bartosinski, R., Daněk, M., Kafka, L., Kohout, L., Sỳkora, J., Bartosiński, R.: The leon3 processor. UTLEON3: Exploring Fine-Grain Multi-Threading in FPGAs pp. 9–14 (2013)
14. Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., et al.: The matter of heartbleed. In: Internet Measurement Conference (IMC) (2014)
15. Harrison, K., Xu, S.: Protecting cryptographic keys from memory disclosure attacks. In: 37th IEEE/IFIP DSN. pp. 137–143. IEEE (2007)
16. Hofemeier, G., Chesebrough, R.: Introduction to Intel AES-NI and Intel Secure Key Instructions. Intel, White Paper **62** (2012)
17. Intel: Software Guard Extensions. `https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf`
18. Intel: Intel Memory Protection Keys. `https://www.kernel.org/doc/html/latest/core-api/protection-keys.html` (2022)
19. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ecdsa). International journal of information security **1**, 36–63 (2001)
20. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: ACM CCS (2003)
21. Koromilas, L., Vasiliadis, G., Athanasopoulos, E., Ioannidis, S.: GRIM: Leveraging GPUs for kernel integrity monitoring. In: RAID 2016. pp. 3–23. Springer (2016)
22. Kounavis, M.E., Kang, X., Grewal, K., Eszenyi, M., Gueron, S., Durham, D.: Encrypting the Internet. ACM SIGCOMM CCR **40**(4), 135–146 (2010)
23. Mao, Y., Chen, H., Zhou, D., Wang, X., Zeldovich, N., Kaashoek, M.F.: Software fault isolation with API integrity and multi-principal modules. In: ACM SOSP (2011)
24. McCamant, S., Morrisett, G.: Evaluating sfi for a cisc architecture. In: USENIX Security Symposium (2006)
25. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In: ACM CCS (2007)
26. Suh, G.E., Clarke, D., Gassend, B., Van Dijk, M., Devadas, S.: AEGIS: Architecture for tamper-evident and tamper-resistant processing. In: ACM ICS (2003)
27. Vasiliadis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: PixelVault: Using GPUs for securing cryptographic operations. In: ACM SIGSAC CCS (2014)
28. Wheeler, D.J., Needham, R.M.: Correction to xtea. Unpublished manuscript, Computer Laboratory, Cambridge University, England **1**(2), 17 (1998)