Hierarchical Firmware-level Security Policy for Industrial Control Systems

Sameer Mankotia*, Daniel Conte de Leon*, Jennifer Johnson-Leung*
*Center for Secure and Dependable Systems, University of Idaho, Moscow, Idaho, USA.
Email: mank8837@vandals.uidaho.edu, dcontedeleon@ieee.org, jenfns@uidaho.edu

***Abstract*—Industrial control systems need strong security and performance guarantees. Current digital systems may be vulnerable to a variety of low-level attacks that exploit common weaknesses such as out-of-bounds read/write, access of resource using incompatible type, and de-serialization of untrusted data. These types of weaknesses appear in the 2024 CWE Top 10 KEV Weaknesses List. In this article, we introduce the binary implementation (BHPol) of the HPol hierarchical security policy framework. BHPol enables the fast firmware-level declaration and enforcement of security polices on-chip. A BHPol enabled digital processor checks each low-level instruction against a predefined and static set of security policies using associative memory. Requests not allowed by the policy are denied.**

***Index Terms*—Power systems security, access control, binary trees, protective relays, real-time systems, hierarchical policies**

I. Introduction

Industrial control systems face significant cybersecurity threats from low-level vulnerabilities that can compromise critical infrastructure operations. The MITRE Top 10 KEV Weaknesses [1] identifies several weaknesses that are currently being exploited (KEV), including, Out-of-Bounds Write (CWE-787), Type Confusion (CWE-843), Use After Free (CWE-416), and Deserialization of Untrusted Data (CWE-502) [1]. These vulnerabilities are particularly dangerous in industrial control systems, where attacks can directly impact physical processes and safety systems.

Traditional security mechanisms are inadequate to fully address these low-level threats in digital industrial control environments. Conventional access control systems like Role-Based Access Control (RBAC) [2] operate at application layers and cannot prevent memory corruption attacks that occur at the instruction level. Furthermore, industrial control systems require deterministic and real-time response, making it difficult to implement software-based security checks with real-time performance assurances.

We propose BHPol (Binary Hierarchical Policy), a novel firmware-level security technology specifically designed to prevent low-level vulnerabilities in industrial control systems based on HPol [3, 4]. BHPol addresses three critical requirements for industrial security systems. First, it provides deterministic timing guarantees for policy-based security decisions through a novel combination of associative memory and a novel hierarchical binary policy representation and querying technology. Second, it operates at the firmware level to intercept and prevent low-level attacks before they can compromise the integrity of the system. Third, it supports hierarchical policies that align well with the structure of digital applications and enables effective policy encoding and enforcement.

II. Related Work

The cybersecurity of Industrial Control Systems (ICS) has become increasingly critical following high-profile attacks on critical infrastructure. Currently, most state-of-the-practice solutions focus on network-level security rather than addressing intrinsic vulnerabilities at the firmware level [5]. Memory safety and other related low-level weaknesses still appear as the largest category (35%) within the top 10 known exploited vulnerabilities [1] and a significant and mostly unaddressed threat vector in ICSs

A. Current Defense Mechanisms and Limitations

Traditional defenses against memory overflows include Stack Canaries (i.e. StackGuard), Address Space Layout Randomization (ASLR), and Data Execution Prevention (i.e. DEP, Non-executable stack). These mechanisms have greatly increased the difficulty of exploiting over the past two decades, but have not solved the fundamental problem. These mechanisms are very high-grained and as a result often by-passable. Control Flow Integrity (CFI) techniques aim to prevent code-reuse attacks, but most are software-based and not implemented at the single instruction level [6]. Policy-based security frameworks such as SELinux implement comprehensive policy frameworks at the kernel level [7] while capability-based security models provide fine-grained access control through unforgettable tokens. These solutions provide insufficient protection against low-level vulnerabilities. Newer processor-based security technologies, such as ARM Trust Zone and Intel SGX, offer better kernel/task or task/task isolation but do not prevent attacks based on program input manipulation or exploits operating in the context of the same task.

B. Current Threat Landscape and Active Exploits

The threat landscape for industrial control systems has evolved significantly, as evidenced by CISA's Known Exploited Vulnerabilities (KEV) Catalog, which maintains an authoritative list of vulnerabilities actively being exploited in the wild [8]. The KEV Catalog serves as a critical resource for understanding which vulnerabilities pose immediate threats to operational technology environments. Recent additions to the

KEV Catalog include several memory corruption vulnerabilities which directly target low-level attack vectors that BHPol is designed to prevent.

The August 2025 release of CISA's "Foundations for OT Cybersecurity: Asset Inventory Guidance for Owners and Operators" [9] emphasizes the critical need for comprehensive visibility into operational technology assets. This guidance, developed in collaboration with the NSA, FBI, EPA, and international partners, highlights that many critical infrastructure organizations lack basic awareness of their OT assets, making them vulnerable to attacks that exploit firmware-level vulnerabilities. The guidance specifically recommends cross-referencing asset inventories with vulnerability databases, including the KEV Catalog, to prioritize remediation efforts for actively exploited vulnerabilities.

C. Gaps in Current Research

We believe that current research literature reveals a significant gap between general-purpose security mechanisms and the specific needs of industrial control systems, particularly the ability to stop attacks at the binary execution level. Current security solutions operate at higher abstraction levels and cannot prevent memory corruption attacks that occur at the firmware and hardware instruction level. Furthermore, existing approaches may introduce unpredictable timing delays that are incompatible with the deterministic requirements of real-time industrial control systems. BHPol was designed to address these critical gaps by providing hardware-level security policy enforcement with hierarchical policy structures. BHPol is designed to prevent malicious operations (defined as not-in-policy) at the instruction level, providing protection against the types of vulnerabilities actively being exploited as documented in the KEV Catalog.

III. How BHPol Works

BHPol implements the hierarchical security policy framework (HPol). HPol [3, 4] uses graph nodes and hierarchies to represent policy entities and their hierarchies and graph paths to represent policies. BHPol is designed to use trees instead of full graphs and enforces policies at the hardware level via associative memory matching and per-instruction enforcement.

In HPol and BHPol, graph nodes represent and organize hierarchically the security elements needed to make a security decision, as illustrated in Figure 1. These hierarchies include the Subject, Action, and Object hierarchies. In HPol and BHPol, policies are defined as an ordered tuple (or path) between nodes in the graph or tree. In HPol a security decision is made by searching the policy graph. In BHPol a security decision is made by matching a query to a table-based and partially ordered representation of the policy set. The number of columns needed on the BHPol policy table is equal to the number of labels in the tuple representing a policy. In the example used in this article, we use a 3-tuple: Subject, Action, Object, but the technology is not limited to a path of length 3.

In the example used in this article, the Subject tree (Who), identifies entities attempting to perform operations. Each entity is assigned a composite label. For each tree, at the root level, we have universal access represented by __.__ indicating *all subjects*, where the underscore symbol is the minimum symbol in the order. The tree may branch into descendants as needed for the corresponding application. For example: User Processes (01.__), System Processes (10.__), and Kernel Processes (11.__). Examples of User Processes could be: configuration handlers (01.01) and data loggers (01.10), these depend on the application's policy needs.

The Action tree (What), defines permitted operations in the system. Starting from __.__ indicating *all actions*. This tree may branch into Read Operations (01.__), Write Operations (10.__), and Execute Operations (11.__). Read operations allow data retrieval without system modification, while write operations enable memory changes or I/O writes.

The Object tree (Where), categorizes resources being operated upon, such as memory objects or I/O. As an example, the hierarchy may progress from Code Segments (01.__) containing executable instructions, to Data Segments (10.__) for normal program operation, and System Areas (11.__) including stack memory and interrupt vectors. Each object category maybe further subdivided (hierarchically nested) into specific memory regions (for both code and data) such as handlers, functions, arrays, structures, buffers, and single values (for example 2 or 4 bytes).

BHPol's binary labeling scheme provides efficient encoding and comparison of hierarchical relationships. Each element receives a unique label where the bit or symbol length grows logarithmically with the number of elements in each category. The hierarchical organization enables inheritance of access permissions where higher-level entities can access resources available to lower-level entities within the same branch, providing a flexible and powerful framework for policy declaration and enforcement. BHPol is designed to be integrated with Hardware-based Memory Management approaches used in the embedded systems as shown in algorithm 1.

A. Policy Evaluation Example

When a Memory or I/O access request occurs, BHPol evaluates the request path through all trees (columns) in the tuple. For example, a configuration handler (01.01) attempting to write data (10.01) to a configuration buffer (10.01) would trace through compatible hierarchical levels. However, a subject (task, function, code section) attempting to take an action (read, write, execute, add, multiply, cast) on objects (memory, I/O, buses) not specified in the policy would be immediately blocked as it violates the established policy.

Table I shows an example of how BHPol would evaluate a configuration handler trying to write data to a buffer. The query (01.01, 10.01, 10.01) represents the handler requesting write access. Policy 3 matches successfully: The Subject label is an exact match (01.01), the Object label is a less-than match (label 10.__ is an ancestor of label 10.01), the Action tree is

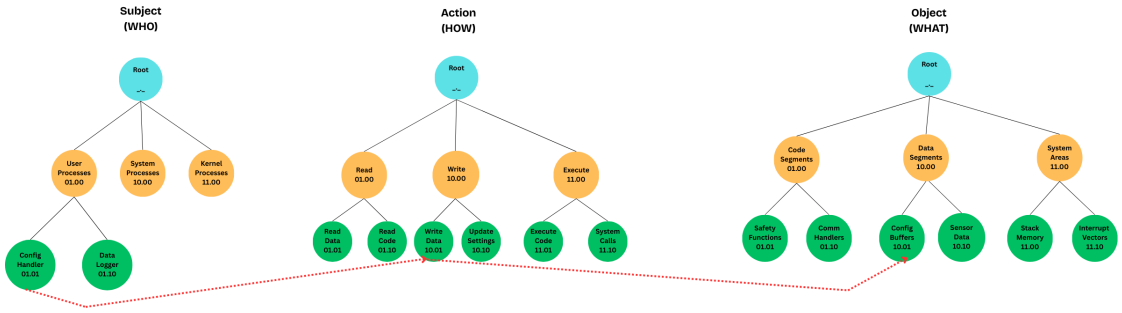

Fig. 1: BHPol Framework: Three Binary Tree Hierarchies for Memory Protection.The highlighted red path demonstrates the policy enforcement flow during memory access operations.

Algorithm 1: BHPol Memory Access Control Algorithm

Memory Access Intercept:
Every resource access triggers BHPol policy check
Extract: Subject label
Extract: Operation type label
Extract: Resource address label

if BHPol_Check(Subject, Action, Object) =< Policy Entry **then**
    Allow access
    Continue normal execution
else
    Generate exception
    Log security violation
    Decide to either terminate or continue offending task
    Alert security monitoring
end if

an exact match (10.01). The evaluation proceeds by comparing a policy query, issued for each single instruction, against each stored policy, in parallel using an associative cache and for each tree/column. The result is that the configuration handler (01.01) is authorized to perform write operations (10.01) on configuration buffers (10.01), because an exact or ancestor match was found. This demonstrates BHPol's efficient binary comparison process, where legitimate operations are quickly authorized while unauthorized access attempts are denied.

IV. Buffer Overflow Protection Case Study

Buffer overflow vulnerabilities represent one of the most persistent threats in digital systems, exploiting the Buffer Copy without Checking Size of Input weakness (CWE-120). Consider a typical vulnerable configuration processing function, as in listing 1.

This implementation contains multiple vulnerabilities exploitable through carefully crafted input data. The `strcpy` function performs unbounded copying that can overflow the destination buffer, while the lack of input validation allows

```c
void process_config_data(char* input_data) {
    char buffer[64];           // Fixed buffer allocation
    strcpy(buffer, input_data); // Unsafe copy operation
    update_system_settings(buffer);
}
```

Listing 1: Example of Vulnerable Configuration Processing

attackers to provide malicious configuration data that corrupts memory and potentially executes arbitrary code. We believe, BHPol provides protection against buffer overflow and other low-level attacks. The protection mechanism operates by classifying memory regions (hierarchically and up to any desired level of granularity, from whole memory to a system word), processes, and operations according to their security requirements and enforcing access control policies for every memory operation.

When a buffer overflow attack occurs against a BHPol-protected system, the security framework intercepts the malicious memory operations through a multi-stage detection and prevention process. The first stage occurs when the malicious payload attempts to overflow the destination buffer. BHPol monitors every memory write operation and compares the target address against allocated buffer boundaries. When `strcpy` attempts to write beyond the 64-byte buffer allocation, the security framework detects the boundary violation and immediately blocks the operation before corruption occurs. The second stage activates when the overflowed return address attempts to redirect program execution to attacker shell code. BHPol classifies stack memory as a restricted system area that should not contain executable code. When the processor attempts to execute instructions from the stack location, the security framework recognizes this as a code injection attack and blocks execution while generating security alerts. The firmware-level implementation operates independently of application software and cannot be bypassed through software vulnerabilities. BHPol security policies would be enforced by hardware memory management units integrated with the processor architecture, providing protection that persists even if the main application code is compromised.

TABLE I: **BHPol Policy Table, Query, and Match Example:** Notice that a policy query may be **equivalent** to an existing policy on the table (associative cache) or may also be a **descendant** (smaller in the partial order of policies), or may not be ordered (not comparable). Thru this novel policy encoding and search BHPol enables the O(1) search on a hierarchy (tree) for each subject, object, action tree.

Policy Index	Subject Label	Subject Match	Object Label	Object Match	Action Label	Action Match	Match Result	Match T/F
0	11.11	-	01.10	-	01.01	-	-	False
1	10.01	-	11.01	-	11.10	-	-	False
2	01.10	-	01.01	-	10.10	-	-	False
3	01.01	≡	10._ _	>	10.01	≡	≡	**True**
4	11.01	-	10.11	-	01.11	-	-	False
5	01.11	-	11.10	-	11.01	-	-	False
6	10.11	-	01.11	-	01.10	-	-	False
QUERY	**01.01**		**10.01**		**10.01**		**Allowed**	

V. Implementation and Evaluation

The performance characteristics are driven by the efficient encoding of the policy hierarchy and the associative memory-based policy search. We have yet to implement a hardware version of BHPol, but the design and objectives are for a policy search cost of O(1) and a few clock cycles. Memory requirements for BHPol policy storage may depend on the level of details needed to fully implement the needed policy for each specific application. Each security policy entry requires a few bytes of storage. We expect that complete policy set for simple industrial control systems will fit within a typical embedded processor cache, in this case dedicated to the policy manager, but have not performed experiments on this space yet.

BHPol was designed to provide consistent timing behavior that can be accurately predicted during system design and verification. We have fully implemented the BHPol in Python along with a testing framework. An example function of the testing framework is shown in listing 2. This listing shows an example of a function to randomly generate a policy. We are currently working on integrating BHPol into our FlexSimArch processor simulator.

The deterministic timing characteristics of BHPol policy evaluation are essential for real-time industrial control applications. Absent of a hardware implementation and corresponding in-hardware measuring experiments we provide here an analysis of why we believe BHPol to be an effective and efficient approach to eliminating low-level weaknesses in ICSs.

Table II presents a quantitative analysis, based on current literature, comparing Associative Cache-based versus other state-of-the-art security mechanisms. This table was created based on performance indicators reported for the major firmware-, software-, and hybrid-based security-focused approaches currently used in modern digital processors. It is important to note that a hardware implementation of BHPol, a corresponding effectiveness and performance evaluation, and formal proofs of model behavior are still future work. The first entry in the table is an estimation based on the fact that BHPol is designed to contain all policy entries in a dedicated associative cache with no software or firmware involvement during the search, thus resulting in an expected query-policy search with O(1) and L1 cache performance characteristics.

```
def generateAndAppendRandomPolicy( self ):
    #
    self.__validateInstanceLevelAssertions();
    #
    # print( "BHPolPolicyMgr: Generating and adding a
         random policy:" , '\n' );
    subjectStr =
         self.__subjectBHPolTree.selectRandomNodeLabel();
    actionStr =
         self.__actionBHPolTree.selectRandomNodeLabel();
    objectStr =
         self.__objectBHPolTree.selectRandomNodeLabel();
    # print(
         "BHPolPolicyMgr.generateAndAppendRandomPolicy:",
         subjectStr, actionStr, objectStr );
    # If the policy entry does not already exists in this
         policy table:
    # -- THE EXACT SAME POLICY IS NOT ALLOWED MORE THAN
         ONCE:
    if ( not
         self.doesPolicyExistInPolicyTableByLabelExactMatch(
          subjectLabelIn=subjectStr,
          objectLabelIn=actionStr,
          actionLabelIn=objectStr ) ):
        self.__appendNewPolicy( subjectStr, actionStr,
             objectStr );
    #
    self.__validateInstanceLevelAssertions();
```

Listing 2: Example Function Used for Test Generation

VI. Future Work

Several critical research directions emerge from the BHPol framework development; Notably, further evaluation, implementation, and validation.

Currently, we have designed and implemented a fully-featured RISC-V (RV32I) focused hardware simulator and compiler and are working on integrating the BHPol approach into the compiler and simulator.

Formal verification of BHPol's security properties requires mathematical proofs of policy correctness and security guarantees. We have completed informal mathematical verification of the model properties, and our next steps are to develop human-readable formal proofs of the framework using Isabelle/HOL. These proofs will support the development of verification tools and methodologies specifically designed for hierarchical policy systems and will contribute to broader security verification for critical systems.

TABLE II: **Performance Comparison with State-of-the-Art Security Methods** [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]

Security Method (Implementation)	Avg. Latency (μs)	Memory Overhead	Energy Overhead	Granularity Level	Hardware Support
Assoc. Cache (BHPol)	**2.3±0.1**	**0.7%**	**1.2%**	**Per instruction**	**Dedicated**
ASLR (Randomization)	15.7±12.4	2.1%	3.8%	Process-level	Software
DEP/NX (Execute Disable)	8.4±6.2	1.3%	2.1%	Page-level	Hardware
Intel CFI (Control Flow)	22.6±18.3	4.7%	6.2%	Function-level	Hardware
ARM Pointer Auth (ARMv8.3+)	12.1±8.7	3.2%	4.5%	Pointer-level	Hardware
Stack Canaries (GCC/Clang)	5.8±2.1	0.9%	1.4%	Function-level	Software
Intel MPX (Memory Protect)	34.7±25.8	8.1%	9.3%	Pointer-level	Hardware
ARM TrustZone (Secure/Normal)	18.9±14.2	6.4%	5.7%	World-level	Hardware
Intel CET (Control-flow)	28.3±21.7	5.8%	7.1%	Branch-level	Hardware

Dynamic policy management also represents an important area for future development. While BHPol's current design assumes static policy configurations, real industrial systems require the ability to adapt security policies in response to changing operational conditions, emergency situations, and maintenance requirements. Research into dynamic xsecure policy update mechanisms that maintain system security while enabling operational flexibility will be essential for practical deployment. We believe that the hierarchical structure of HPol and the symbol-based labeling of policy entities is well-suited for enabling policy merging (via tree composition), with relabeling if needed. However, we have not fully investigated this approach yet.

VII. Limitations

BHPol was designed to provide strong protection against memory safety violations and other low-level attacks but has limitations.

The current threat model does not consider high-level application compromise such as impersonation or insider threats; Nor considers low-level out-of-chip weaknesses such as side-channel or timing attacks.

The system currently assumes static policy configurations that cannot adapt to changing operational needs without manual intervention, making dynamic real-time updates challenging.

Success depends on accurate hierarchical policy setup; Configuration errors could block legitimate operations or permit unauthorized access, requiring extensive testing.

BHPol's limitation from its approach to represent hierarchical categorization and labeling of Memory and I/O resources (Object tree) is that each level of the policy tree represents a memory region size, for example, 16 bytes for a node (and level) and 16x16 bytes for its direct ancestor (and level), as aligned with the memory addressing scheme. This may lead to memory segmentation in a system using real memory addressing. If using page-based memory addressing, the labeling approach may lead to internal segmentation. On the contrary, a segment-based virtual memory addressing scheme, aligned with BHPol's Object labeling approach, would reduce or eliminate fragmentation.

Performance claims remain theoretical as BHPol hasn't been implemented on actual embedded hardware, potentially revealing unexpected overhead or compatibility issues during real deployment. Additionally, formal proofs of the model properties are not yet available.

Since BHPol is designed to fully integrate within a digital processor, integration with existing industrial control systems, after extensive testing and evaluation are completed, will require the gradual replacement of ICS devices. Hence, the deployment times would be limited by device replacement cycles leading to long deployment timelines. Hence, it should not be considered the only mitigation technique used in a given system, but instead, a long-term technology and approach for the mitigation of weaknesses that have remained in our digital systems for many decades.

VIII. Conclusion

We believe that BHPol represents a novel approach in protecting industrial control systems from attacks exploiting low-level weaknesses as described in the MITRE 2024 CWE Top 10 KEV Weaknesses List Insights article [1]. By implementing hierarchical security policy checks directly in hardware, BHPol is designed to provide protection against low-level attacks and memory safety issues that existing security techniques (software, firmware, and hardware) fail to effectively mitigate.

BHPol uses an innovative hierarchical approach for representing policy entities and policies, inherited from HPol. BHPol introduces an innovative technique for encoding this hierarchical structure in an efficient manner (tree->table) that enables O(1) query/policy matching when using an associative cache containing the policy. The presented example shows how BHPol would prevent multi-step attacks targeting memory corruption by intercepting threats at each instruction execution creating layered defense that aims to ensure a secure state at all times.

IX. Acknowledgment

The authors would like to thank Schweitzer Engineering Laboratories (SEL) and the State of Idaho for funding this research. The opinions expressed in this article are solely the authors'.

References

[1] MITRE Corporation, "2024 cwe top 25 most dangerous software weaknesses - kev list," MITRE

Corporation, 2024, common Weakness Enumeration. [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_kev_list.html

[2] R. S. Sandhu, "Role-based access control," in *Advances in Computers*. Elsevier, 1998. [Online]. Available: https://doi.org/10.1016/S0065-2458(08)60206-5

[3] D. Conte de Leon, M. Brown, A. Jillepalli, A. Stalick, and J. Alves-Foss, "High-level and formal router policy verification," *Journal of Computing Sciences in Colleges*, vol. 33, no. 1, pp. 1–12, Oct. 2017.

[4] A. H. Alkhorem, D. Conte de Leon, A. A. Jillepalli, and J. Song, "Formalizing permission to delegate and delegation with policy interaction," *Sensors*, vol. 25, no. 16, p. 4915, aug 2025. [Online]. Available: https://doi.org/10.3390/s25164915

[5] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE, may 2013, pp. 48–62. [Online]. Available: https://doi.org/10.1109/SP.2013.13

[6] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys*, vol. 50, no. 1, pp. 1–33, apr 2017. [Online]. Available: https://doi.org/10.1145/3054924

[7] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, NSA and NAI Labs. Berkeley, CA, USA: USENIX Association, 2001, pp. 29–42.

[8] Cybersecurity and Infrastructure Security Agency, "Known exploited vulnerabilities catalog," U.S. Department of Homeland Security, 2021, continuously updated catalog of vulnerabilities exploited in the wild. [Online]. Available: https://www.cisa.gov/known-exploited-vulnerabilities-catalog

[9] e. a. CISA, "Foundations for ot cybersecurity: Asset inventory guidance for owners and operators," U.S. Department of Homeland Security, Joint Guidance Document, aug 2025, tLP:CLEAR.

[10] H. Marco-Gisbert and I. Ripoll Ripoll, "Address space layout randomization next generation," *Applied Sciences*, vol. 9, no. 14, p. 2928, jul 2019. [Online]. Available: https://doi.org/10.3390/app9142928

[11] L. Binosi, G. Barzasi, M. Carminati, S. Zanero, and M. Polino, "The illusion of randomness: An empirical analysis of address space layout randomization implementations," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, dec 2024, pp. 1360–1374. [Online]. Available: https://doi.org/10.1145/3658644.3690239

[12] Arm Limited, "Understand arm pointer authentication: Pointer authentication on arm," Online learning resource, 2024, aRM Learning Path. [Online]. Available: https://learn.arm.com/learning-paths/servers-and-cloud-computing/pac/pac/

[13] Z. Cai, J. Zhu, W. Shen, Y. Yang, R. Chang, Y. Wang, J. Li, and K. Ren, "Demystifying pointer authentication on apple m1," in *Proceedings of the 33rd USENIX Security Symposium*, ser. USENIX Security '24. Berkeley, CA, USA: USENIX Association, 2024.

[14] C. Cowan, S. Beattie, R. Finnin Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting systems from stack smashing attacks with stackguard," in *Proceedings of the Linux Expo*. Department of Computer Science and Engineering: Oregon Graduate Institute of Science & Technology, 1998, contact: crispin@cse.ogi.edu. [Online]. Available: http://www.cse.ogi.edu/DISC/projects/immunix

[15] K. Kim, J.-N. Kim, and S. Lee, "Stackguard+: Interoperable alternative to canary-based protection of stack smashing," *Electronics Letters*, oct 2024. [Online]. Available: https://doi.org/10.1049/ell2.13310

[16] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–30, jun 2018. [Online]. Available: https://doi.org/10.1145/3224423

[17] M. Cole and A. Prakash, "Simplex: Repurposing intel memory protection extensions for information hiding," sep 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2009.06490

[18] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys*, vol. 51, no. 6, pp. 1–36, jan 2019. [Online]. Available: https://doi.org/10.1145/3291047

[19] Q. Zhu, Q. Chen, Y. Liu, Z. Akhtar, and K. Siddique, "Investigating trustzone: A comprehensive analysis," *Security and Communication Networks*, apr 2023, open Access. [Online]. Available: https://doi.org/10.1155/2023/7369634

[20] Intel Corporation, *A technical look at Intel control-flow enforcement technology*, Intel Developer Documentation, 2021.

[21] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2019.

[22] Microsoft Corporation, *Data execution prevention - Win32 apps*, Microsoft Learn Documentation, 2021.

[23] Linux Kernel Documentation Team, "Intel(r) memory protection extensions (mpx)," Online documentation, The Linux Kernel Organization, 2019, linux Kernel Documentation, version 5.4. [Online]. Available: https://www.kernel.org/doc/html/v5.4/x86/intel_mpx.html