Hardening Intel® Trusted eXecution Technology and Intel® Boot Guard

Security White Paper

July 2024

Revision 0.21

Authors:

Foundational Software Technologies
Intel Product Assurance and Security

Document Number: 824411

Contents

Tables

Revision History

Document Number	Revision Number	Description	Revision Date
824411	0.1	• Initial draft	October 2023
	0.11	• Add platform support table • Clean up unneeded references	April 2024
	0.12	• Add diagrams and standardized names	June 2024
	0.13	• Clean up additional references	June 2024
	0.2	• Accuracy improvements	July 2024
	0.21	• Accuracy improvements	July 2024

§§

1 Introduction

When the Trusted Computing Group (TCG) began its operation in the late 90's, it was in a bid to promote trust in the personal computing platform. Intel, a member of the group, has implemented its specification for Dynamic Root of Trust for Measurement (D-RTM), in the form of Intel® Trusted eXecution Technology (TXT). A dynamic root of trust enables measurement and attestation of platform state as the system boots into a measured launch environment (MLE). Later, Intel also implemented a Static Root of Trust for Measurement (S-RTM), in the form of Intel® Boot Guard (BtG). Together, Intel® Trusted eXecution Technology (TXT) and Intel® Boot Guard (BtG) are part of Intel® Hardware Shield offering for Intel's vPro® capable platforms. Security is the primary commercial and corporate design value proposition for these features, allowing a computing platform operator to prevent tampering of the platform root of trust and establish a trusted and protected environment for operating systems and software.

Intel's implementations are partly based on low-level privileged firmware known as Intel Authenticated Code Modules (ACMs). This firmware is authenticated by microcode and then loaded into an internal cache for execution. Intel develops this ACM firmware and integrates it with platform reference BIOS, while continually engaging in effective high-assurance secure embedded software development, in close collaboration with the Intel Product Assurance and Security (IPAS) group. Given its privileged nature and its security premise, it is imperative for such firmware to remain robust, both in terms of quality and against adversarial attempts to circumvent protections.

As experience shows, developing bug-free code is hard. Therefore, the team devised and executed on a firmware hardening roadmap, accounting for the unique needs of the firmware at hand, to deliver multiple incremental security improvements. Though hardening alone cannot guarantee the absence of bugs, it can serve to reduce their exploitability. To name a few, these transparent enhancements include Libsafe Containment (also known as Secure Memory Copy), Shadow Stack and Shadow Pointers for control flow integrity or redundancy, Post Quantum Cryptography and Data XOR Code (also known as Memory Isolation). These firmware enhancements address classes of vulnerabilities (e.g., classic buffer overflows) in a minimally intrusive manner, to collectively reduce Intel platforms' attack surface and business risk. This whitepaper describes these hardening measures, alongside references for context.

§§

2 Security Scope

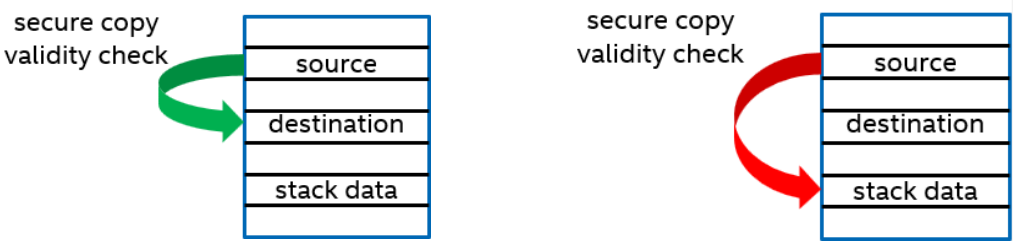
Security is a notoriously elusive goal. In addition to strictly adhering to Security Development Lifecycle (SDL) practices, Intel works to remediate internally discovered and externally submitted product security incident response tickets (PSIRTs) and close security gaps. A useful approach has been to root out classes of vulnerabilities altogether en masse, as opposed to treating individual bugs. Intel chose an incremental hardening strategy, to reduce the impact and severity of as yet undiscovered security vulnerabilities. Hardening applies to any platform supporting Intel Trusted Execution Technology (TXT) and Intel Boot Guard. This includes Client and Server markets, catering to the largest OEMs.

3 Libsafe Containment

Overview

ACMs protect control metadata during copy operations, by walking the call stack as part of vetted copy functions, ensuring copy operations do not clobber important data.

This technique (also known by the name "Secure Memory Copy") leverages the codebase's use of vetted copy functions, in order to place within these copy functions a dedicated security check, that determines whether the copy destination is valid within a stack frame or not. If the check fails, then the firmware proceeds with graceful failure, thwarting an attack. The use of existing copy functions ensures consolidated security checks without littering the codebase, while preserving stack integrity. The technique is inspired by libsafe.



Simplified Illustration: validity check allows a source buffer to be copied to a legitimate destination (left pane), but not to a sensitive destination (right pane)

References

A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 251-262, San Jose, CA, June 2000. USENIX. [LINK]

4 Shadow Stack

Overview

A shadow stack mechanism for control flow integrity addresses the notorious buffer overflow technique that has plagued implementations over the past four decades. A shadow stack adds control redundancy, by maintaining a separate copy of the call stack. This separate copy may reside in the data segment, adjacent to the regular stack, in hardware, or other place in memory. The location will depend on the constraints of the particular firmware. Pushes and pops to the call stack correspond to symmetric pushes and pops to the shadow stack. When popping the call stack, the shadow stack can be cross-referenced for consistency, and an inequality indicates a violation, to be dealt with appropriately.

ACMs leverage the compiler to introduce instrumentation in the form of calls at function entry and exit, injecting custom security logic that maintains a shadow copy of control metadata. Upon entering a function, an entry hook is called to store a copy of the return address on the shadow stack. Upon exiting a function, an exit hook is called to perform a security check that determines whether the two copies diverge. If the check fails, then firmware proceeds with graceful failure, thwarting the attack. The use of custom injected security logic ensures consolidated security checks and a transparent mitigation.

References

K. Samelson and F. L. Bauer. 1960. Sequential formula translation. Commun. ACM 3, 2 (Feb. 1960), 76–83.

Bauer F.L. (2002) From the Stack Principle to ALGOL. In: Broy M., Denert E. (eds) Software Pioneers. Springer, Berlin, Heidelberg.

Smashing the stack for fun and profit, by Aleph One, Phrack Magazine Issue 49 https://web.archive.org/http://phrack.org/issues/49/14.html

H. Shacham (2007) The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM conference on Computer and communications security (CCS '07). Association for Computing Machinery, New York, NY, USA, 552–561. DOI: https://doi.org/10.1145/1315245.1315313

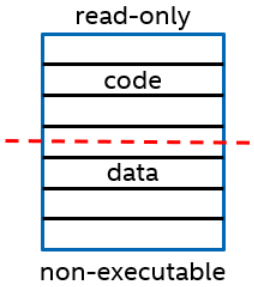
L. Szekeres, M. Payer, T. Wei and D. Song, "SoK: Eternal War in Memory," 2013 IEEE Symposium on Security and Privacy, 2013, pp. 48-62, doi: 10.1109/SP.2013.13.

N. Burow, X. Zhang and M. Payer, "SoK: Shining Light on Shadow Stacks," 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 985-999, doi: 10.1109/SP.2019.00076.

5 Data XOR Code

Overview

ACMs implement W^X, to avoid an unrestricted von Neumann machine. Within ACMs, code is not writable, and data is not executable. In order to achieve this separation, ACMs enable x86/x64 paging modes early in the execution, and define code pages and data pages, each with suitable page table attributes, such that memory isolation will be enforced by hardware upon every memory access. If writing code or fetching data is attempted, then an ACM will abort gracefully with an error message.



Simplified Illustration: memory is separated into code and data, the former being executable but not modifiable, the latter being modifiable but not executable

References

J. VON NEUMANN, "FIRST DRAFT OF A REPORT ON THE EDVAC," IN IEEE ANNALS OF THE HISTORY OF COMPUTING, VOL. 15, NO. 4, PP. 27-75, 1993
HTTP://PEOPLE.CSAIL.MIT.EDU/BROOKS/IDOCS/FIRSTDRAFT.PDF

INTEL (2024) INTEL 64 AND IA-32 ARCHITECTURES SOFTWARE DEVELOPER'S MANUAL: COMBINED VOLUMES (1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, AND 4)
HTTPS://WWW.INTEL.COM/CONTENT/WWW/US/EN/DEVELOPER/ARTICLES/TECHNICAL/INTEL-SDM.HTML

PAX TEAM, PAX NON-EXECUTABLE PAGES DESIGN AND IMPLEMENTATION
HTTPS://PAX.GRSECURITY.NET/DOCS/NOEXEC.TXT

WINDOWS XP SERVICE PACK 2 DATA EXECUTION PREVENTION
HTTPS://H10032.WWW1.HP.COM/CTG/MANUAL/C00387685.PDF

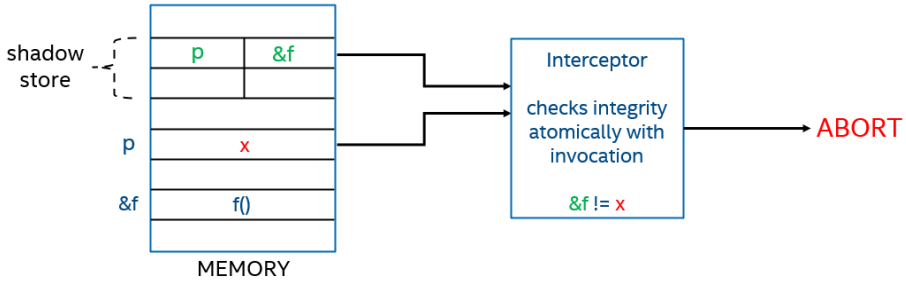
6 Shadow Pointer

Overview

ACM embedded firmware is hardened by protecting function pointers, using a form of forward edge control flow integrity which we dub "shadow pointers", a specialized notion of a generalized shadow memory. With this technique, function pointers are copied into a shadow store, to be cross-referenced upon function invocation. In such a manner, malicious function pointer modifications are caught and gracefully handled.

Protection is achieved by pre-processing the code with a transpiler – an LLVM-based source transformation tool for instrumenting pointer assignment and dereference. Upon function pointer assignment, a hook is inserted for updating shadow memory. Upon function pointer dereference, a hook is inserted for checking that the pointer has not changed. The instrumented code is subsequently compiled with the protection.

Naturally, the protection afforded by the technique is limited to the window between function pointer assignment and invocation, and cannot defend against an adversary overwriting both the original function pointer and the shadow function pointer.



Simplified Illustration: integrity check upon function pointer invocation determines that a function pointer has been corrupted relative to its shadow copy and gracefully aborts – the discrepancy is that the pointer 'p' no longer points to 'f' as it should

References

CFIMon: Detecting Violation of CFI using Performance Counters DSN'12
Protecting Function Pointers in Binary (FPGate) CCS'13
Code-Pointer Integrity USENIX OSDI'14
Missing the Point(er): On the Effectiveness of Code Pointer Integrity
Getting the Point(er): On the Feasibility of Attacks on Code-Pointer Integrity
Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM USENIX'14
Practical Context-Sensitive CFI (PathArmor) CCS'15
Per-Input Control Flow Integrity (PICFI) CCS'15
CCFI: Cryptographically Enforced Control Flow Integrity CCS'15
Capturing 0day Exploits with PERFectly Placed Hardware Traps BH'16
Efficient Protection of Path-Sensitive Control Security (PittyPat) USENIX'17
Rest In Protection: A Kernel-level Approach to Mitigate RIP Tampering ICISSP'17
Security Analysis of Processor ISA for Enforcing Control-Flow Integrity HASP'19

7 Post-Quantum Cryptography

Overview

Literature suggests that quantum computers may have capabilities to break or weaken classic cryptographic systems approximately by the year 2030. To raise the bar against quantum computing attacks on classic cryptographic systems, standards are being developed to enhance security.

For post quantum cryptography (PQC) resilient digital signatures, ACM is supporting Leighton-Micali Signatures (LMS) for all manifests. The ACM signature structure format will maintain its familiar layout, with a minor change to the version bit to indicate hybrid signature support.

This optional hybrid approach for post quantum cryptography (PQC) resilient digital signatures is recommended when the public key hash is stored in silicon and backward compatibility is needed until all OEM customers of ACM are ready to follow the recommendation to use LMS only. For example, if necessary, the Boot Guard Key Manifest (KM) may use LMS combined with currently supported legacy schemes RSASSA or ECDSA.

Because LMS private keys are limited in the number of signatures they can produce before the key is permanently disabled, careful analysis was needed to confirm the number of signatures can satisfy all Boot Guard manifest requirements.

References

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, "RECOMMENDATIONS FOR STATEFUL HASH-BASED SIGNATURE SCHEMES," OCTOBER 2020. [ONLINE]. AVAILABLE: HTTPS://NVLPUBS.NIST.GOV/NISTPUBS/SPECIALPUBLICATIONS/NIST.SP.800-208.PDF.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, "POST-QUANTUM CRYPTOGRAPHY WORKSHOPS AND TIMELINE," 8 APRIL 2024. [ONLINE]. AVAILABLE: HTTPS://CSRC.NIST.GOV/PROJECTS/POST-QUANTUM-CRYPTOGRAPHY/WORKSHOPS-AND-TIMELINE

8 Security Engineering

Good security starts with good engineering. Intel continues to enhance its software development and validation practices, as demonstrated by:

- **Requirements Engineering**: Product requirements are recorded and traceable to ensure that adequate validation is completed. Monthly checkpoints with stakeholders are held to ensure requirements are addressed with consensus and to track progress on requirement validation.
- **Architecture**: Architectural design is established based on requirements. Unique constraints for hardening the architecture are captured in requirement collaterals.
- **Design and Implementation**: A detailed design is developed based on the requirements and architecture. That detailed design is used to describe and implement the product. For each hardening measure, the detailed design documentation includes security concerns and mitigations.
- **Unit Testing**: Software modules are verified to ensure compliance with requirements and verify the correctness and robustness of the solution. Unit Tests also confirm that mitigations defend against known security issues.
- **Build Process and Test Automation**: A consistent build process has been established, including integration and testing. Some functional and systems testing can run without manual intervention, so automated test scripts are used where appropriate. Test cases are categorized based on their purpose and may be executed with different cadences.
- **Peer Reviews**: Periodic checkpoints ensure that all stakeholders review and provide feedback on feature designs, with findings tracked to closure. Collaborative code reviews occur on a weekly basis. Patches are not checked in until they get approval by at least two deciders and one developer. Regular hackathons are conducted by a red team.
- **Policy Compliance**: Best known methods are followed to ensure compliance with security, privacy, and legal policies. This includes following Intel's defined Security Development Lifecycle process.
- **Quality Assurance**: Independent assessment of the quality of work is done against pre-defined standards.
- **Project Management**: Project activities, schedule, and budget are planned, monitored, and adjusted.
- **Risk Management**: Project risks are tracked and managed, following best known methods to reduce risk. Reported vulnerabilities help the team pursue data-driven countermeasures.

§§

9 Conclusion

Intel® TXT and Intel® Boot Guard represent major building blocks for a safe computing environment. Security is continually and incrementally improved via new defense in depth and anti-exploitations techniques adopted into Intel Authenticated Code Modules. Intel is committed to continue working on hardening Intel technologies and delivering state-of-the-art solutions to enable and enhance users' secure-computing experience.

10 Appendix

Table 1 – Platform Support for Intel® ACM security enhancements

	Client	Server
Libsafe Containment	Supported	Supported
Shadow Stack	Supported	Supported
Data XOR Code	Supported	Supported
Shadow Pointer	Supported	Supported
Post-Quantum Cryptography	Supported	Supported

§§