TRUSTED® COMPUTING GROUP

REFERENCE

TCG Runtime Integrity Preservation in Mobile Devices

Family "2.0"
Level 00 Revision 106
November 11, 2019

Contact: admin@trustedcomputinggroup.org

PUBLISHED

DISCLAIMERS, NOTICES, AND LICENSE TERMS

ACKNOWLEDGEMENTS

The TCG wishes to thank all those who contributed to this reference document. This document builds on work done in other TCG work groups, including Infrastructure, Storage, Trusted Mobility Solutions, and Trusted Platform Module.

Special thanks to the following current and former members of the Mobile Platform WG who contributed to this document:

Name	Affiliation
Wael Ibrahim (MPWG co-chair)	American Express
Andre Rein	Andre Rein
Bo Bjerrum	Bo Bjerrum
Carlin Covey (editor)	Carlin Covey
Amy Nelson	Dell, Inc
Michael Eckel	Fraunhofer Institute for Secure Information Technology (SIT)
Graeme Proudler	Graeme Proudler
Tom Laffey	Hewlett Packard Enterprise
Steve Hanna	Infineon Technologies
Mitul Shah	Intel Corporation
Alec Brusilovsky	InterDigital Communications, LLC
Ira McDonald (co-editor)	Ira McDonald
Kathleen McGill (MPWG co-chair)	Johns Hopkins University, Applied Physics Lab
Guy Fedorkow	Juniper Networks, Inc
Charles Schmidt	The MITRE Corporation
Jessica Fitzgerald-McKay	United States Government

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Scope and Audience

NIST Special Publication 800-164 (Draft) Guidelines on Hardware-Rooted Security in Mobile Devices (Draft) states: "Mobile devices should implement the following three mobile security capabilities to address the challenges with mobile device security: device integrity, isolation, and protected storage." This TCG Runtime Integrity for Mobile Devices (RIP) document addresses the first of these security capabilities by recommending practices and mechanisms that are intended to preserve the integrity of the critical portions of the runtime state of mobile devices. This entire document addresses security considerations for mobile devices including many of the security topics discussed in [2].

Mobile device manufacturers and enterprise administrators can establish security policies for runtime mobile device integrity that identify the portions of the runtime state that are considered critical and which actors are authorized to modify the mobile device runtime state. RIP mechanisms within the mobile device can prevent or detect and remediate runtime state modifications made by unauthorized actors.

This document provides recommendations for improvements in the security of mobile devices and mechanisms to allow mobile devices to maintain enhanced security during operation. The recommendations in this document are targeted at designers, developers, and implementers of trusted computing technologies in mobile devices.

TCG Mobile Reference Architecture [1] defines secure boot for mobile devices (which ensures that the mobile device starts in a known state) and includes diagrams of a variety of common mobile device architectures. Naturally, some aspects of the mobile device state are expected to change as software executes or writes to data areas. However, portions of the mobile device state can still be checked for authorization to execute and for integrity.

1.2 Informative References

1. Trusted Computing Group, TPM 2.0 Mobile Reference Architecture v2r142, December 2014, https://trustedcomputinggroup.org/wp-content/uploads/TPM-2-0-Mobile-Reference-Architecture-v2-r142-Specification_FINAL2.pdf

2. IETF, Guidelines for Writing RFC Text on Security Considerations, RFC 3552, July 2003, https://tools.ietf.org/html/rfc3552

3. Trusted Computing Group, Storage Security Subsystem Class: Opal, Version 2.01 Final, Revision 1.00, August 2015, https://trustedcomputinggroup.org/wp-content/uploads/TCG_Storage-Opal_SSC_v2.01_rev1.00.pdf

4. The Clang Team, Control Flow Integrity – Clang 7 Documentation, April 2018, https://clang.llvm.org/docs/ControlFlowIntegrity

5. C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, Enforcing Forward-Edge Control-Flow Integrity, April 2014, https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf

6. PaX Team, Rap: Rip Rop, April 2018, https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf

7. Microsoft, (Enable Control Flow Guard), April 2018, https://msdn.microsoft.com/en-us/library/dn919635.aspx

8. Qualcomm, Qualcomm releases whitepaper detailing pointer authentication on ARMv8.3, January 2017, https://www.qualcomm.com/news/onq/2017/01/10/qualcomm-releases-whitepaper-detailing-pointer-authentication-armv83

9. M. Castro, M. Costa, and T. Harris, Securing Software by Enforcing Data-flow Integrity, in the Proceedings of the 7th Symposium of Operating System Design and Implementation, November 2006, https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/2006-osdi.pdf

10. J. Kong, C. Zou, H. Zhou, Improving Software security via runtime instruction-level taint checking, in Proceedings of the 1st workshop on Architectural and system support for improving software dependability, October 2006, http://www.cs.ucf.edu/~czou/research/ASID06.pdf

11. P. Kohli, Coarse-grained Dynamic Taint Analysis for Defeating Control and Non-control Data Attacks, July 2009, https://arxiv.org/abs/0906.4481

12. T. Jim, J. G. Morrisett, D. Grossman, M. Hicks, J. Cheney, Y. Wang, Cyclone: A Safe Dialect of C, in Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, 2002, trevorjim.com/papers/usenix2002.pdf

13. S. Nagarakatte, J. Zhao, M.K. Martin, S. Zdancewic, SoftBound: Highly Compatible and Complete Spatial Memory Safety for C, ACM SIGPLAN Notices, v.44 n.6, June 2009, https://repository.upenn.edu/cgi/viewcontent.cgi?article=1941&context=cis_reports

14. G.C. Necula, J. Condit, M. Harren, S. McPeak, W. Weimer, CCured: type-safe retrofitting of legacy software, ACM Transactions on Programming Languages and Systems (TOPLAS), v.27 n.3, p.477-526, May 2005, https://dl.acm.org/citation.cfm?id=1065892

15. J. Devietti, C. Blundell, M. K. Martin, S. Zdancewic, HardBound: Architectural Support for Spatial Safety of the C Programming Language, ACM SIGARCH Computer Architecture News, v.36 n.1, March 2008, https://www.cis.upenn.edu/acg/papers/asplos08_hardbound.pdf

16. R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S.W. Moore, S. J. Murdoch, R. Norton, S. Son, and H. Xia, Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6), April 2017, http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-907.pdf

17. IANA, TLS Parameters Registry. (Standard cryptographic algorithms), https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml

18. J. Butler, Black Hat: Windows 2004 - DKOM (Direct Kernel Object Manipulation), 2004, https://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf

19. A. Prakash, E. Venkataramani, H. Yin, and Z. Lin, Manipulating semantic values in kernel data structures: Attack assessments and implications, in 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 12, June 2013, http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6575344

20. M. Castro, M. Costa, T. Harris, Securing software by enforcing data-flow integrity, In Proceedings of USENIX OSDI, 2006, https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/2006-osdi.pdf

21. NIST, FIPS 199 Standards for Security Categorization of Federal Information and Information Systems, February 2004, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.199.pdf

22. Trusted Computing Group, TCG Glossary, http://www.trustedcomputinggroup.org/developers/glossary

23. T. Bletsch, Code-reuse attacks: new frontiers and defenses, Ph.D. Dissertation, 2011, https://dl.acm.org/citation.cfm?id=2338075

24. F. Wang, Understanding Code-Reuse Attacks and Reducing Attack Surface, October 2017, https://medium.com/mit-security-seminar/understanding-code-reuse-attacks-and-reducing-attack-surface-d7349a507dc7

25. A. Pendergrass and K. McGill, LKIM: The Linux Kernel Integrity Measurer, Johns Hopkins APL Technical Digest, vol. 32, no. 2, 2013, https://www.jhuapl.edu/techdigest/TD/td3202/32_02-Pendergrass-McGill.pdf

26. NIST Computer Research Security Center, Glossary, https://csrc.nist.gov/glossary/term/SP

27. Trusted Computing Group, Trusted Platform Module Library Family 2.0 Level 00 Revision 1.38, September 2016, https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0.zip

1.3 Definitions

Term	Definition	Source
Integrity Preservation	The process and techniques for ensuring that the integrity of the device is preserved while the device is running. Includes enforcement, assessment and remediation of integrity.	FIPS 199 Standards for Security Categorization of Federal Information and Information Systems [21]
Service Processor	A processor with isolated resources that configures resources used by the main application processor(s). The service processor does not execute user-supplied software. The service processor may provide other management services in additional to configuration.	NIST Computer Research Security Center [26]
Trust	Trust is the expectation that a device will behave in a particular manner for a specific purpose.	TCG Glossary [22]

2 RUNTIME INTEGRITY PRESERVATION CONCEPTS

2.1 Mobile Device Integrity Goals

The goal of runtime integrity preservation is to ensure that the device continues to behave in an expected manner following a successful secure boot. Since it is difficult to characterize the expected behavior of the device and compare expected behavior to actual behavior, this document relies on the following assumptions:

- The device executes a secure boot process, which is defined in [1].
- If critical portions of the device state are preserved at runtime, the device will behave in a manner expected by the device manufacturer.

2.1.1 Integrity of Reference Measurements

Integrity assessment mechanisms compute a measurement (e.g., a secure hash) over software and/or static data objects in memory, and compare the result of these measurements against secure reference metrics to determine whether the software or data objects have been corrupted.

In order for integrity assessment mechanisms to detect corruption of data objects it is essential that the authenticity of the reference metrics be verified, and that their integrity be protected thereafter. One method to authenticate reference metrics is to verify a cryptographic signature over the reference metrics. If reference metrics are embedded within a software distribution package, then the stored reference metrics will be authenticated at boot time along with the software itself.

Reference metrics must match the software image as it exists at runtime, but the software could have been transformed when it was loaded into memory. A solution to this challenge is to have the RIP mechanism calculate fresh metrics after the software has been transformed, but before any other software is allowed to execute.

If reference metrics are stored in RAM, then the reference metrics themselves could be altered. This alteration could be prevented by appropriate countermeasures. For example, the reference metrics could be stored instead in protected storage, such as flash, kernel address space, or a TPM. Reference metrics could also be protected by cryptographic integrity checking via digital signatures or HMAC/CMAC computations.

2.1.2 Identity of Protected Objects

Cryptographic integrity protection (e.g. via digital signature or HMAC/CMAC) can be used to verify that an object has not been altered since the reference metric was calculated. However, this protection cannot guarantee that the software is in fact referencing the correct object. For example, cryptographically protected return address pointers could have been maliciously modified between function calls. A possible solution to this challenge is to embed a function identifier in the protected return address and then verify the function identifier before the return address is dereferenced.

The cryptographic integrity protection assurance of data objects depends upon proper access control for the protection keys. Only those entities that can be trusted to provide authentic data should be allowed to apply the cryptographic protection. Only trusted entities should be allowed to perform the action that is authorized when the protected data is authenticated.

2.1.3 Freshness of Integrity Assessment

The probability of a successful integrity attack increases over time. Therefore, the highest assurance of mobile device integrity immediately follows an integrity assessment. Calculating integrity metrics consumes mobile device resources, including power, bus bandwidth, and possibly processor cycles. This mobile device resource consumption can be mitigated by reducing the frequency of integrity assessment. The freshness of the integrity assessment is a matter of system policy. The need for integrity assessment freshness could be more critical for certain mobile devices or certain sensitive operations.

2.2 Mobile Device Integrity Preservation

A device transitions through several phases to reach an operational state. Operational state is defined as the condition in which the device is ready for its intended purpose. The pre-boot phase begins with initial firmware execution on the device, which might perform various hardware health checks and hardware configuration actions. The boot phase can invoke device support firmware and the bootloader. The bootloader loads runtime system software such as a hypervisor or Operating System (OS) kernel. From this point onward, the runtime system software controls the state of the device and further software execution.

2.2.1 Pre-boot Integrity

Pre-boot integrity refers to the integrity of the mobile device prior to the boot of one or more of the main runtime processors. This phase includes the period when the SoC is completely powered down (except perhaps a coin cell battery to run a small amount of logic, such as the real time clock and monotonic counters), the period when the SoC is in an offline mode (network interfaces are not active and service processor is not running), and the period when a service processor is running but the main processor is not.

The integrity of the runtime system might be dependent on the pre-boot integrity of certain elements of the system, such as the configuration of non-volatile storage in ROM, flash, fuses, or state-retention registers. Unauthorized modification of these elements prior to boot can subvert the security of the boot process.

2.2.2 Boot-time Integrity

At power-on-reset, the mobile device begins executing firmware from a Boot ROM or secure flash memory. The firmware is integrity protected by the immutability of the ROM or secure flash memory. Using an integrity-protected signature verification key, the firmware verifies the signature over the boot image that will be executed next. If the signature is correct, the boot ROM firmware initiates execution of that boot image. That boot image then verifies the signature over one or more additional blocks of software using the same public key or other integrity-protected public keys. If any of these checks fail, the mobile device enters a remediation mode, or returns to the reset state [1]

2.2.3 Load-time Integrity

Before software is executed on a mobile device, it must be loaded properly. Usually, the software is available in persistent storage, such as a hard-drive or flash-chip. The loading of the system software is initiated by boot software. In the Secure Boot process, the integrity and authenticity of loaded software is verified before it is allowed to execute. Once the OS has been loaded, it typically manages the loading of application software. The OS verifies the integrity of the application software and transforms it into a ready-to-run state (e.g., resolving dynamic library links).

2.2.4 Run-time Integrity

Runtime integrity preservation protects security-critical portions of the mobile device state during execution. Runtime integrity preservation requires appropriate integrity measurement and assessment of security-critical components. Identifying the relevant security-critical portions of the mobile device is an important aspect of runtime integrity preservation. In addition to software, data, such as policies, credentials and configuration files, can affect the security of the mobile device at runtime.

Due to the large number of potential attacks, no one solution by itself can ensure system integrity. Multiple technologies are necessary to ensure resilience despite the spectrum of runtime integrity attacks. These technologies can enable the assessment of integrity violations, allow integrity recovery following a detected integrity violation, or increase resistance to integrity violations. The TCG Cyber Resilient Technologies workgroup focuses on supporting system resilience.

3 RUNTIME INTEGRITY PRESERVATION (RIP) MECHANISMS

The following subsections describe a variety of runtime integrity preservation mechanisms. These mechanisms are examples of existing techniques, but they are not exhaustive. Any runtime integrity preservation techniques that follow the recommendations in Section 5 are considered valid implementation choices.

Some mobile devices incorporate a service processor that provides management services to the main processor cores, such as initialization, integrity checking, mobile device management, and system update. The service processor operates in an isolated environment and manages the configuration of the other processor(s). As a result, the service processor is a natural candidate for performing runtime integrity checks on the other main processors' software images. Some runtime integrity preservation mechanisms can leverage a service processor if available. However, since the service processor often can read and modify most of the mobile device resources, the integrity of the service processor itself is critical.

Static integrity mechanisms preserve the integrity of the mobile device's static state, e.g. the configuration files, software and other static data. Dynamic integrity mechanisms verify that the dynamic state of the mobile device falls within expected parameters. Static and dynamic runtime integrity protection mechanisms are complementary. Static integrity assessment and remediation ensure the integrity of software and additional information embedded in the software image that can be used for dynamic integrity mechanisms. Dynamic integrity is necessary to assess and remediate attempts to modify data or control state.

3.1 Static Integrity Mechanisms

Static integrity mechanisms contribute to the preservation of the integrity of critical portions of the mobile device state.

3.1.1 Static Integrity Enforcement

Some RIP techniques provide static integrity protection of data. In other words, the techniques explicitly prevent the compromise of protected components.

3.1.1.1 Immutable Mobile Device Resources

One technique for enforcing integrity at runtime is to employ immutable mobile device resources, e.g., hardware or Read-Only Memory. Strict immutability is the safest way to preserve the integrity of mobile device resources. One approach to prevent modifications is to store software in ROM, however this mechanism is only practical for certain specialized software, such as boot software or integrity assessment software. A more flexible, but more complex approach, is to constrain the modifications that are made to software storage areas. For instance, software can be loaded into RAM at boot time under the control of integrity-protected software, and then, the ability to write to RAM would be disabled until the next power cycle.

However, often even nominally "immutable" resources are actually mutable under certain conditions. For one example, ROM can have patch hardware that allows certain subsets of ROM memory locations to be modified to correct firmware bugs. For a second example, the state of a fuse bank can be buffered in flipflops, and the state of these flipflops can be alterable, even if the fuse bank is not. In each of these cases, it is the responsibility of access control mechanisms to prevent unauthorized modifications of these resources and thereby preserve the inherent integrity. The nature of these access control mechanisms is out of scope for this document.

3.1.1.2 Bulk Storage Integrity

Another technique for enforcing integrity at runtime is through the use of an integrity-protected bulk storage device. The TCG Storage Work Group focuses on specifications for security services on dedicated storage systems. One objective of the TCG Storage Work Group is to develop specifications and practices for defining the same security services across dedicated storage controller interfaces, including but not limited to ATA, Serial ATA, SCSI, FibreChannel, USB Storage, IEEE 1394, Network Attached Storage (TCP/IP), NVM Express, and iSCSI. Storage systems include disk drives, removable media drives, flash storage, and multiple storage device systems. The TCG Storage Security Subsystem Class: Opal Specification [3] provides detail on the use of a Storage Device in a trusted platform (including mobile devices), including storage integrity use cases and capabilities.

3.1.1.3 Stored Data Protection

A common technique for ensuring the integrity of data and software in memory is to implement a hardware lock that, when set, prevents any write operations to a specific area of RAM. This technique can be used to ensure that data or software loaded by the boot software remains immutable during mobile device execution until the next power cycle.

A similar hardware lock could be used to prevent read operations. This technique can be used to allow boot software to read secret data from ROM, but prevent software that runs after boot time from obtaining the secret data. A use case for this read-lock mechanism is protection of a private or symmetric cryptographic key used during mobile device provisioning.

Another hardware lock could be used to protect data encrypted with the mobile device storage key. This technique allows boot software to encrypt data that cannot be later decrypted once the boot software has set the hardware lock. Further, if the data is encrypted with an algorithm that appends a cryptographic signature, then later software cannot encrypt data that spoofs the boot software.

3.1.1.4 Data Execution Prevention

Data Execution Prevention (DEP) is a hardware mechanism for enforcing an execution permission policy for memory pages. When the OS kernel loads software into a memory page the kernel sets a hardware flag that identifies the page as executable. If a processor tries to fetch instructions from a memory page that is not flagged as executable, an exception is generated and the program is terminated. This mechanism prevents integrity compromise by misusing valid portions of an image.

3.1.1.5 Canary Values

The use of canary values is a software-based bounds-checking mechanism for preventing the exploitation of buffer overflows. Typically, this mechanism is implemented within the compiler. The compiler places a canary value in-between buffers and control data, and adds check routines that are invoked before accessing the control-data. If an inserted canary value changed from the expected value, the program is terminated.

3.1.2 Integrity Assessment

Runtime integrity assessment is the process of determining whether the security-critical portions of the system state are consistent with their expected values. In many cases, this is the state that was established at boot time. This boot-time state includes the system configuration and the software and data that are loaded during the boot process. In order to detect changes from the expected security-critical boot-time state, the RIP mechanism must have reliable measurements of this security-critical boot-time state.

Note that certain variations of the boot-time state are expected and permissible:

- Data areas will change state during the execution of the system software and application software
- The operating system can load additional applications after boot-time.
- Changes to the hardware configuration can occur as flash cards are inserted or removed during runtime.

In most cases integrity assessment will be limited to those portions of the state that are expected to remain constant following boot. In some cases, the mobile device can operate in different modes, and portions of the mobile device state can be dependent on the mode. The integrity assessment mechanism would need to be aware of the mobile device's current mode and use the appropriate integrity reference metrics.

If the integrity assessment detects a deviation from the expected state, one or more remediation actions should be initiated.

The following subsections describe examples of integrity assessment metrics. Any of these techniques can be applied in combination. That is, one mechanism might be used for integrity checking software that performs integrity checking of other software and data, resulting in a hierarchy of integrity checking mechanisms.

3.1.2.1 Protected Copy of Image

If there is a copy of the software/data image in protected storage (e.g. ROM, access-controlled flash memory), the image in RAM can be periodically compared byte-by-byte with the protected copy during runtime. Similarly, subsets of the image can be compared this way.

3.1.2.2 Protected Hash of Image

A hash can be computed over a trusted image or critical portions and stored as an integrity-protected reference metric in protected storage. During runtime, the RIP mechanism can compute the hash over the integrity-protected portions of the runtime image and compare the newly-calculated hash with a protected reference hash.

3.1.2.3 HMAC/CMAC Portions of Image

Critical software and data can have HMAC/CMACs embedded in the image. These HMAC/CMACs could be generated at compile time or at link time. Periodically, the RIP mechanism could randomly select a portion of the image, compute the HMAC/CMAC over that portion, and compare the computed HMAC/CMAC with the embedded HMAC/CMAC.

3.1.2.4 Watchdog Timers

Watchdog timers can be used to support integrity assessment (and also remediation). Once started, a watchdog timer increments or decrements at regular intervals until the count reaches a predefined value, at which point the timer expires. When the timer expires, some action is initiated to assess integrity. In this way, a watchdog timer can detect and limit the impact of integrity compromise. However, it cannot prevent compromise.

There are several cases in which a watchdog timer is particular useful:

1. A procedure with a known execution time: The timer would start at the initiation of the procedure and expire at the predicted time when execution is expected to terminate. This would detect “hung” or “runaway” conditions.
2. Software response to a hardware event: The watchdog timer would start when the hardware event was detected, and the software that responds to the event would turn off the timer before it expires. The hardware event could be an interrupt request initiated by a security sensor (e.g. voltage out of range, temperature out of range), or it could be a software-initiated action such as a request to a hardware security module. If the expected response did not occur before the timer expired, software would be notified.

When the watchdog timer expires, the remediation action is a matter of mobile device security policy. If the mobile device has a service processor or other isolated processing element, it might be possible to perform remediation actions on a subset of the mobile device state.

3.1.3 Integrity Remediation

After an integrity compromise has been detected, it is generally unsafe to continue normal mobile device operation until the compromise has been remedied. The appropriate remediation actions for a given scenario are a matter for the mobile device security policy. Possible remediation actions include rebooting the mobile device or attempting to restore the expected state of the mobile device while it continues to operate. The RIP remediation mechanisms ensure that the remediation actions specified by the mobile device security policy are initiated, and that any failures in remediation actions are handled appropriately. This policy can be very simple (e.g. one remediation action in all cases) or it can be complex (e.g. different remediation actions for different cases). The following are examples of remediation mechanisms.

3.1.3.1 Reboot

A simple remediation technique is a full reboot of the mobile device without any attempt to preserve the context of the mobile device. The integrity of the mobile device is restored via the normal secure boot process. In mobile devices that incorporate two or more processors it might be possible to reboot the processor that experienced an

integrity failure without rebooting the mobile device. For example, a discrete TPM [27] could be left running to preserve data objects in secure storage.

3.1.3.2 Reload a Subset of Software

If it is possible to identify the particular software or data object whose integrity has been compromised, then that data object could be restored without rebooting the mobile device. For example, if integrity is assessed in a hierarchical fashion, an integrity failure can be detected at higher levels of the hierarchy while lower foundational levels are uncorrupted. In these cases, the remediation action can address higher-level objects.

3.1.3.3 Special Case: Inter-device Transaction Remediation

More complex remediation actions might try to preserve the context of certain operations that were active at the time that the integrity compromise was detected. For instance, if two devices have entered into a transaction (e.g. a sales transaction), both devices should be notified if the integrity state of either device is compromised during the transaction. In that event, application-specific software on both devices can take actions to back out of the transaction.

3.2 Dynamic Integrity Mechanisms

Many of the concepts that apply to static integrity enforcement also apply to dynamic integrity enforcement. However, dynamic integrity mechanisms can only address the integrity of the dynamic state of the mobile device during runtime.

3.2.1 Control Flow Integrity

Control Flow Integrity (CFI) mechanisms protect running software from unintended or malicious modifications that would subvert its control flow [4]. These modifications are known as control-data attacks – often called Code Reuse Attacks. The control flow is subverted by exploiting vulnerabilities inside the software (e.g. stack or heap-based buffer overflow [12] or underflow) that allow the manipulation of control-data such as a branch target (i.e. a function pointer). This manipulation enables the adversary to execute/reuse code or portions of code in unintended ways [23, 24]. Consequently, the goal of CFI is to prevent the execution of unintended/maliciously-altered execution flows and, thus, prevent control-data attacks [13]. This goal can be achieved by preventing the initial modification itself [7] or by preventing the execution of the malicious modification. Typically, the types of CFI mechanisms include hardware-enforced (e.g. shadow-stacks, pointer-authentication), software enforced by applying self-checking code (e.g. branch-target tracking), or a combination of both.

Conceptually, CFI is divided into forward-edge [5] and backward-edge [6] protection policies. Both involve an analysis of the software's Control Flow Graph (CFG) to distinguish between valid and invalid branch-targets. Typically, forward-edge protection is implemented via compiler-generated checks inserted into the code. If a control flow transition is found that deviates from the intended CFG, the program is terminated. Forward-edge protection is currently implemented in some products.

Backward-edge transitions follow specified call conventions and processor behavior and are thus usually suitable for hardware enforcement techniques. One mechanism for backward-edge protection is Pointer Authentication [8]. When the processor executes a function call, a cryptographic signature, called a PAC (Pointer Authentication Code), is appended to the return address pointer. The PAC is then verified whenever a protected pointer is dereferenced. If the verification is successful, then program execution continues, else the processor traps to an integrity error handler.

Control-data attacks are almost always only the initial attack to a system, enabling further compromise. One major use-case for these attacks is to disable memory protection technologies such as Data Execution Prevention (DEP). This can facilitate easier exploitation and more persistent modification of the system. Once DEP is disabled, the code can be corrupted and remain active on a system until the afflicted system component is terminated. In particular, in cases where the OS kernel is attacked, the system is compromised until rebooted. For this reason, additional measures should be applied to prevent the execution of maliciously modified code or to detect that a system or component was compromised, in order to trigger additional remediation actions. There already are advanced integrity

assessment solutions that detect integrity compromises of OS code and even OS configuration when the code itself is not corrupted [25].

3.2.2 Data Flow Integrity

Similar to CFI, Data Flow Integrity (DFI) protects software execution flows and enforces a runtime-policy that prevents malicious attacks leveraging of the program's CFG. These attacks are called non-control data attacks, and their major distinction from control-data attacks is that they limit themselves to the modification of data only. No code pointers are modified. In general, different security-critical data structures are subject to non-control data attacks, such as configuration data, user input data, user identity data, and decision-making data. Passwords and private keys, randomized values or system call parameters are also potential targets for non-control-data attacks.

Systematic manipulation of the non-control data can lead to confidentiality leaks, privilege escalation attacks, or arbitrary code execution. These outcomes are made possible by intelligent manipulation of data used in different control-structures such as if-statements, loops or assignments. In contrast to control-data attacks, only legitimate execution flows are used.

There have been different non-control-data attacks in the past. Root-kits that target the OS kernel often rely on systematic manipulation of data-structures. One prominent example in this regard is process-hiding, which refers to making a malicious user-space process invisible or seemingly benign to the user and administrator of a system to avoid its detection [18, 19].

DFI protects against non-control data attack exploitation techniques during runtime. Two countermeasures are available for use in implementing a DFI policy; Data Flow Graph DFG [9] and Dynamic Taint Analysis (DTA) [10, 11]. These two approaches are similar to CFI methods, but in this case the data flow is monitored instead of the control flow. In principle, both approaches model legitimate data transitions and detect deviation from these models. Both solutions induce high overhead. At time of writing, these approaches are active areas of research.

3.2.3 Memory Corruption Prevention

Control-data and non-control-data attacks always rely on an initial exploitation to subvert or leverage the control-flow of software. CFI or DFI policies do not prevent the initial exploitation of a vulnerability, e.g. overflow a buffer to write arbitrary values to memory; instead they employ mechanisms to detect or prevent execution after the initial attack. Memory Corruption Prevention establishes policies that prevent the initial attack, i.e., corruption of relevant control or non-control-data structures. Consequently, without the initial memory corruption, the actual attack does not occur.

Unmanaged programming languages, such as C or C++, that do not enforce strict bounds-checking on memory operations and pointer-dereferencing policies, are particularly susceptible to memory corruption. There are various software or hardware countermeasures that can reduce these vulnerabilities in unmanaged programming languages. Typically, hardware implementations provide significantly higher performance and security guarantees. Software-based techniques originate from concepts for type-safe C [14]. For instance, several existing products implement fat-pointers that, in addition to a normal address pointer, contain meta-data used for enforcing bounds-checking that provides full or partial spatial memory safety. Hardware-based countermeasures are also available, for instance Hardbound [15] and, most recently, CHERI [16]. CHERI implements strict bounds-checking on the basis of introduced capabilities and further enhances the protection of arbitrary data-structures in memory.

4 RUNTIME INTEGRITY PRESERVATION RECOMMENDATIONS

This section lists recommendations for RIP. The tags (e.g., “def_auth_state”) used below are for convenience of the reader and allow backward references from a given recommendation/rationale to a previous one. This list of recommendations is designed to consider all of the runtime integrity mechanisms described above in section 3.

4.1 Recommendations for RIP Security Policy

- Recommendation [def_auth_state]: The manufacturer of a mobile device should provide an unambiguous definition of valid mobile device runtime states and the entities authorized to update these state definitions.
    - Rationale [def_auth_state]: The secure boot process ensures that the mobile device is initially in an authorized runtime state. However, following secure boot, the mobile device will modify data areas as software executes. Some mobile devices might allow firmware over-the-air updates that modify the system software. Most mobile devices allow application software to be loaded and executed at runtime. The RIP mechanisms must take into account which of these state modifications are permissible, so the mechanisms can determine when the mobile device’s runtime integrity has been compromised.
- Recommendation [resp_policy]: The manufacturer of a secure mobile device should specify the appropriate responses to detection of each runtime integrity compromise in a RIP Security Policy.
    - Rationale [resp_policy]: If an RIP mechanism detects a compromise of the mobile device’s runtime integrity, then the RIP mechanism can initiate remediation actions. For example, in some cases, these actions might involve requesting that the operating system terminate a process. In other cases, the mobile device can be reset.

4.2 Recommendations for Pre-boot Integrity

- Recommendation [rip_preboot]: The mobile device architecture and implementation should protect RIP mechanisms both prior to and during boot either by:
    - Inherent integrity protection (e.g. implemented in ROM or other immutable hardware); or
    - Explicit integrity protection (e.g. measured boot with RoT-based validation).
    - Rationale [rip_preboot]: If the RIP mechanisms are not integrity protected prior to and during boot, then there cannot be any assurance that the mobile device state is integrity protected. The RIP mechanisms can be implemented hierarchically, with each RIP mechanism layer protected by the layer below. This hierarchy is protected by an inherently integrity protected layer.

4.3 Recommendations for Runtime Integrity Enforcement

- Recommendation [preserve_policy]: The mobile device should incorporate mechanisms that preserve the integrity of the mobile device firmware and software in accordance with mobile device security policy.
    - Rationale [preserve_policy]: Although the initial state of the mobile device is ensured via Secure Boot, various malicious attacks or software or hardware flaws could later compromise the mobile device’s runtime state. Runtime integrity preservation mechanisms provide countermeasures that protect mobile device integrity following Secure Boot.
- Recommendation [preserve_rots]: The mobile device should protect the runtime integrity of RoTs.
    - Rationale [preserve_rots]: See Rationale [preserve_policy].
- Recommendation [protect_pe]: RIP mechanisms outside the Protected Environment (PE) should protect the integrity of the entire PE firmware as well as code, runtime data, and trusted applications in the PE.
    - Rationale [protect_pe]: The mobile device can include RIP mechanisms outside the PE (for example hardware or microcode) to enforce the integrity of the PE.

- Recommendation [protect_rip]: The mobile device architecture and implementation should protect the integrity of the RIP mechanisms from untrusted software or hardware.
    - Rationale [protect_rip]: See Rationale [protect_pe].

4.4 Recommendations for Runtime Integrity Assessment

- Recommendation [rip_assessment]: The mobile device architecture and implementation should use RIP mechanisms to assess the mobile device firmware, software, and configuration state.
    - Rationale [rip_assessment]: See Rationale [preserve_policy]
- Recommendation [std_crypto]: Runtime integrity measurement mechanisms should use international standard cryptographic algorithms [17] and should avoid the use of proprietary cryptographic algorithms when collecting integrity measurements.
    - Rationale [std_crypto]: Proprietary cryptographic algorithms often have unrecognized flaws that can allow an attacker to circumvent the integrity assessment. International cryptographic standards have been widely analyzed, and are far less likely to have such flaws.
- Recommendation [ref_metric_authenticity]: The RIP mechanisms should verify the authenticity of all reference metrics.
    - Rationale [ref_metric_authenticity]: The RIP detection mechanisms can be compromised if an attacker could substitute reference metrics that match altered software.
- Recommendation [ref_metric_protect]: The RIP mechanisms should protect the integrity of all runtime integrity reference metrics at all times.
    - Rationale [ref_metric_protect]: See Rationale [ref_metric_authenticity].
- Recommendation [rip_audit_log]: The RIP mechanisms should record the time and date of any modification to RIP-protected code, data, or keys in an audit log.
    - Rationale [rip_audit_log]: The RIP Security Policy can allow portions of the integrity-protected mobile device state to be updated at runtime, but the update process could be subverted to install malicious software/firmware or configuration data. Recording all updates in an audit log allows for forensic analysis.
- Recommendation [fresh_ref_metrics]: The RIP mechanisms should calculate fresh reference metrics (except where precomputed hashes apply to unchanged protected regions) during Secure Boot or as soon as possible thereafter.
    - Rationale [fresh_ref_metrics]: See Rationale [ref_metric_authenticity]
- Recommendation [protect_confidentiality]: All RIP mechanisms implemented by the mobile device should prevent the exposure of confidential data during the execution of RIP mechanisms.
    - Rationale [protect_confidentiality]: The RIP assessment mechanisms could access confidential data (e.g. keys) that would normally be inaccessible. This recommendation prevents the vulnerabilities caused by RIP mechanisms themselves.
- Recommendation [protect_measurements]: RIP assessment measurements should be integrity-protected until verified against reference metrics.
    - Rationale [protect_measurements]: See Rationale [ref_metric_authenticity]
- Recommendation [policy_compromise_log]: If the RIP mechanisms detect an integrity failure in hardware or software used for Security Policy enforcement, then the RIP mechanisms should log this failure in access-protected nonvolatile storage.

- Rationale [policy_enforcement_log]: An integrity failure in the support hardware or software for Security Policy enforcement could be an indication that the Security Policy is no longer being enforced, and the mobile device is compromised.

- Recommendation [on_demand_int_assessment]: The RIP mechanisms should support triggering of an on-demand integrity assessment.
  - Rationale [on_demand_int_assessment]: A fresh integrity assessment could be necessary to increase security assurance for security-critical operations such as financial transactions or to respond to an externally requested attestation.

4.5 Recommendations for Runtime Integrity Remediation

- Recommendation [initiate_remediation]: If the RIP remediation mechanism receives an alert from the RIP assessment mechanism, then the RIP remediation mechanism should initiate the remediation action specified by the RIP Security Policy.
  - Rationale [initiate_remediation]: The RIP remediation mechanism is the ideal entity to receive RIP assessment mechanism alerts and directly initiate remediation actions.
- Recommendation [alert_system]: If the RIP remediation mechanism receives an alert from the RIP assessment mechanism, then the RIP remediation mechanism should send an alert to the affected system software.
  - Rationale [alert_system]: In some cases the RIP mechanism cannot initiate the remediation action by itself, so the RIP mechanism needs to forward the alert to accomplish remediation.

4.6 Recommendations for Control Flow Integrity Preservation

- Recommendation [forward_edge]: The mobile device architecture and implementation should enforce a forward-edge protection policy that checks all control flow transfers during execution.
  - Rationale [forward_edge]: The execution control flow can be altered by manipulating function address pointers during runtime, so it is insufficient to enforce integrity of the static code image.
- Recommendation [backward_edge]. The mobile device architecture and implementation should enforce a backward-edge protection policy that checks all control flow transfers implicitly during execution.
  - Rationale [backward_edge]: The execution control flow can be altered by manipulating return addresses during runtime, so it is insufficient to enforce integrity of the static code image.

4.7 Recommendations for Data Flow Integrity Preservation

- Recommendation [data_flow_integrity]: The mobile device architecture and implementation should host software that enforces a data flow integrity policy [20] by checking data flow transfers during execution.
  - Rationale [data_flow_integrity]: The data flow can be altered by manipulating data and data pointers during runtime. Compile-time analysis can insert runtime integrity protection code inline in software images.

5 PERFORMANCE OF RIP MECHANISMS

Assessing the integrity of data objects in mutable storage (e.g. RAM) at runtime is an active process that consumes resources, e.g. energy (battery power), processor cycles, and memory bus cycles. If these resources are shared with other software processes, then an integrity assessment slows down other software processes. There is an inherent tradeoff between latency and resource consumption. If more frequent integrity assessments are performed, then this slows down other software processes. If less frequent integrity assessments are performed, then there will be greater latency in detecting an integrity failure. Naturally the impact of this tradeoff is situation dependent. For instance, when a mobile device is engaged in a high-value activity such as a financial transaction reduced detection latency is more desirable than reduced resource consumption. If the mobile device is simply streaming a movie, then the opposite tradeoff might be favored.

5.1 Dedicated RIP Hardware

RIP mechanisms can rely on hardware that is dedicated to their functions. Such hardware can offer advantages in terms of performance and integrity of the RIP implementation. Performance could be improved because integrity assessment does not require processing within main processors, leaving more cycles available for other activities. Integrity could be improved because the RIP hardware is immutable and hence impervious to malicious modification.

For example, a dedicated RIP hardware mechanism can read selected portions of memory, compute a hash over that data, and compare the newly computed hash result to a stored reference metric. If there is a mismatch, the RIP hardware mechanism can invoke the remediation actions dictated by RIP Security Policy.

