Protecting Data on Smartphones and Tablets from Memory Attacks

Patrick Colp[†], Jiawen Zhang[‡], James Gleeson[‡], Sahil Suneja[‡],
Eyal de Lara[‡], Himanshu Raj[*], Stefan Saroiu[*], and Alec Wolman[*]

[†]University of British Columbia, [‡]University of Toronto, and [*]Microsoft Research

Abstract

Smartphones and tablets are easily lost or stolen. This makes them susceptible to an inexpensive class of memory attacks, such as cold-boot attacks, using a bus monitor to observe the memory bus, and DMA attacks. This paper describes Sentry, a system that allows applications and OS components to store their code and data on the System-on-Chip (SoC) rather than in DRAM. We use ARM-specific mechanisms originally designed for embedded systems, but still present in today's mobile devices, to protect applications and OS subsystems from memory attacks.

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: Hardware/software interfaces; D.4.1 [*Operating Systems*]: Process Management; D.4.6 [*Operating Systems*]: Security and Protection

Keywords encrypted memory, encrypted RAM, AES, cold boot, bus monitoring, DMA attack, ARM, cache, iRAM, Android, Nexus, Tegra

1. Introduction

Smartphones and tablets are more easily lost or stolen than desktops or even laptops. This ease of loss makes enterprises and device owners concerned about the security of the data stored on their devices. Many enterprises today mandate that employee laptops use disk encryption, such as TrueCrypt [51] or BitLocker [30], to protect "data at rest". With such systems, the data on disk is always encrypted with a key protected by a password or PIN that the user must enter at boot time. If a device is lost, an attacker cannot retrieve the data without knowing the encryption password. Because laptops are often shutdown or hibernating, such systems provide adequate safety when dealing with lost devices.

In contrast, smartphones and tablets are rarely powered off. Instead, when the screen is off, these devices are in a sleep state where RAM is continually refreshed. While sleeping, these devices periodically check for incoming phone calls or application notifications. When a device is lost, an attacker only needs to press a button to force the device to immediately resume execution. Encrypting data at rest is therefore less useful for smartphones and tablets, as their data can always be decrypted and loaded to RAM with the press of a button.

Instead, smartphones and tablets offer *PIN-unlock*: a user must enter a PIN to unlock her device when the device is idle for more than a short period of time (e.g., 15 minutes). To prevent brute-force attacks on the PIN, many smartphones enter a deep-lock state if an incorrect PIN is entered a few times in a row. A key limitation of PIN-unlock is that even when the device is locked, unencrypted data resides in RAM on the stolen device. This allows attackers to subject lost smartphones and tablets to physical attacks. There are different avenues to read secrets from RAM, such as cold boot attacks [25, 31], attaching a bus monitor to monitor data transfers between the CPU and system RAM [18, 21, 24], or mounting DMA attacks [7, 9, 36]. For example, Müller and Spreitzenbarth describe how to mount a cold boot attack on Android smart phones using only a household freezer, a USB cable and a laptop [31]. Using their tool named Frost, they were able to recover recent emails, photos, and visited web sites from physical RAM.

This paper demonstrates that the ARM *System-on-Chip (SoC)* architecture used by today's smartphones and tablets is amenable to a new security approach – storing users' sensitive data on the ARM SoC rather than in DRAM. By storing the secrets on the SoC, we make physical attacks more difficult to mount because they must target the SoC to retrieve secrets, which is much more expensive. The ARM SoC is already equipped with low-capacity storage that can be used as an adequate alternative to traditional off-the-SoC DRAM. Its primary usage is to store small amounts of data closer to the CPU than DRAM. This storage's presence is (arguably) due to the role ARM plays in embedded systems, which often require both fast and predictable performance. Bringing data closer to the CPU helps these systems meet their real-time performance needs.

This paper presents Sentry, a system that protects against DRAM attacks by leveraging on-SoC storage mechanisms originally intended for realtime predictable performance. Sentry can bootstrap additional secure storage by safely encrypting regions of memory much larger than the capacity of the ARM SoC. Sentry starts with the observation that *protecting sensitive data on an unlocked device offers little value because an attacker can access the data simply by using the device's UI*. Instead, Sentry encrypts the memory of sensitive applications and OS subsystems when a mobile device transitions to a screen-locked state. When the user unlocks the device, Sentry decrypts memory on demand to reduce user-perceived resume latency and to save power.

Sentry supports two alternatives to storing secrets in DRAM. First, Sentry supports iRAM, a small amount of internal SRAM on the SoC, whose primary usage is storing the runtime state of the firmware of the platform's peripherals. Second, Sentry supports

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..
Copyright © 2015 ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/2694344.2694380

cache locking, an old feature of ARM platforms, to "pin" data in the cache and prevent it from being written back to DRAM. Current ARM platforms used by smartphones and tablets are equipped with shared L2 caches with much larger sizes than iRAM.

To enable sensitive applications to run in the background while the device remains locked, Sentry leverages AES_On_SoC. Unlike traditional AES, AES_On_SoC safely encrypts and decrypts without leaking any sensitive code and data to DRAM. This new implementation carefully manages the sensitive encryption/decryption state to eliminate the possibility of accidental leaks to DRAM due to procedure calls (e.g., by passing parameters on the stack) or to context switches. Sentry uses AES_On_SoC and fine-grained software control over the L2 cache to secure the state of sensitive applications running in the background. When an application starts running in the background, Sentry transparently reads its encrypted memory pages from DRAM, stores them on the ARM SoC, and decrypts them. Similarly, pages are encrypted when written back to DRAM. Sentry operates with little or no overhead when the device is unlocked, and can securely run background applications while the device is locked, albeit at lower performance.

We implement Sentry on a Nexus 4 smartphone and on an NVidia Tegra 3 development board. On the Nexus 4, Sentry only uses iRAM to store secrets on the SoC, while on the NVidia board, Sentry can use both iRAM and the L2 cache. We have firmware access to the NVidia Tegra 3 board that allows us to enable L2 cache locking (this feature is often disabled by firmware). We use Sentry to secure a number of Android applications (Twitter, Google Maps, Contacts, MP3), and *dm-crypt*, a generic Linux block-level encryption module. Our evaluation shows that Sentry's performance and energy overheads are modest, and its mechanisms do not affect the rest of the system's performance in a significant manner. On a daily basis, Sentry consumes about 2% of a device's battery life to protect an application assuming the user unlocks the device 150 times a day.

2. Sentry

Sentry secures mobile devices against DRAM attacks by leveraging the limited amount of secure storage available on the ARM SoC to safely encrypt much larger regions of DRAM.

Main Observation. Sentry builds on the observation that securing data in memory while a device is unlocked provides limited protection because any user can access sensitive data by simply interacting with the device's UI. Instead, Sentry secures memory state only while a device's screen is locked.

Sentry encrypts the memory pages of all sensitive applications when the device starts transitioning to a screen-locked state. When the device resumes and is successfully unlocked, Sentry decrypts memory pages on demand as sensitive applications start to run. The encryption/decryption keys are only stored on the ARM SoC when the device is locked, and not in DRAM.

While this encrypt-on-lock and decrypt-on-unlock cycle improves security, it has one major drawback – it does not support background computation. Modern smartphones need to perform limited background computations (e.g., receiving notifications, providing calendar alerts) while the screen is locked. To enable this, Sentry runs sensitive background applications in a mode that guarantees their decrypted state is always confined to the SoC.

This approach makes it possible for Sentry to operate with little or no overhead when the device is unlocked, while simultaneously enabling it to run background applications while the device is locked, albeit at lower performance.

Implementing Sentry requires solving the following three technical challenges:

In Scope Attacks	Out of Scope Attacks
cold boot	software attacks (malware)
bus monitoring	physical side-channel attacks
DMA attacks	code-injection
	JTAG attacks
	sophisticated physical attacks (e.g., using electron microscope)

Table 1. Summary of our threat model.

1. On-SoC Storage: Identifying a location on the SoC where sensitive state (e.g., encryption keys, cleartext pages of sensitive background applications) can be stored and accessed while running background applications. All this state must be preserved between lock and unlock cycles.

2. Encrypted DRAM: To enable background operation while locked, Sentry requires a way to transparently run background applications whose memory pages are always encrypted in DRAM. To run these processes, Sentry uses the virtual memory system to load referenced memory pages onto the SoC, and then decrypt them in place. Similarly, Sentry encrypts them before paging them out of the SoC and back to DRAM.

3. On-SoC Encryption: To bootstap its security guarantees, Sentry requires the encryption/decryption routines to never leak any sensitive information outside the SoC. For example, a generic cryptographic library is insufficient because its function calls may place their arguments on the stack in DRAM, and as a result they would violate Sentry's guarantees.

After we describe our threat model, we then address each of the above challenges in turn.

3. Threat Model

This section describes three classes of memory attacks within our threat model, and additional threats that fall outside the scope of our paper. Table 1 summarizes these threats.

3.1 In-Scope Threats

We are concerned with memory attacks that aim to steal secrets by reading the memory of a stolen/lost smartphone or tablet.

Cold Boot Attacks. Cold boot attacks exploit the data remanence effect of RAM to read secrets stored in memory [25]. There are two ways to mount cold boot attacks. One way is for an attacker to physically remove the RAM modules from a victim device and insert them into a compromised device which outputs the RAM's content. The bolting of DRAM into the motherboard and the growing popularity of package-on-package (PoP), where memory is vertically stacked on top of processor, makes this approach ineffective on today's mobile phones.

A more practical possibility is to reboot a stolen device and boot into an attacker-controlled OS that outputs the memory contents [11]. A typical way to mount this attack is by *reflashing* a device. This requires a short power-disconnect (i.e., tapping a RESET button). Müller and Spreitzenbarth used this technique to mount successful cold boot attacks on Android smart phones [31]. On phones with a locked bootloader, they were able to recover recent emails, photos, and visited web sites from physical RAM dumps. Moreover, on devices with an unlocked bootloader, they recovered encryption keys from RAM and decrypted the user partition[1].

[1] Decrypting the user partition on locked devices is not possible because the unlocking process requires device reflashing which wipes all user data.

Bus Monitoring Attacks. An attacker could attach a bus monitoring tool [18, 21] to the memory bus and wait for the CPU to request sensitive data over the memory bus. For example, a simple reboot ensures that any secrets (e.g., disk encryption keys) are loaded into the CPU as they are needed to start decrypting the disk volume upon startup.

Bus monitoring attacks present a different danger than cold boot. Bus monitoring enables a class of side-channel attacks based on memory access patterns. Consider a sensitive application for which an attacker can monitor its access patterns to its internal state. For example, AES implementations use tables of precomputed values (e.g., the exponentiation of 2 in a particular field, such as $GF(2^8)$) to speed up their computation. While the tables themselves are not secret, the order in which the table entries are accessed can reveal secret information. Previous work has shown fast ways to break AES if its state access patterns are known [50]. Bus monitoring attacks can reveal these access patterns, whereas cold boot attacks cannot.

To the best of our knowledge, all previous work on protecting secrets against memory attacks (on x86 platforms) is susceptible to this attack [32, 33, 41]. Addressing this attack is nontrivial because the above schemes can only protect small quantities of data, such as one set of encryption keys. Protecting the entire state of the encryption/decryption code and data makes the secret state's size grow several fold. Such sizes are beyond the protection capabilities of previous solutions. Our related work section (Section 9) provides a more in-depth discussion of these issues.

DMA Attacks. An attacker could program a DMA-capable peripheral to manipulate the DMA controller and read arbitrary memory regions. This class of attacks is often referred to as *Firewire attacks*, although they can exploit many other DMA interfaces, including PCMCIA, PCI Express, or even the more recent Thunderbolt interface. A DMA interface allows a peripheral to read memory contents without any cooperation from the processor or the OS. DMA attacks are successful even when the mobile device is PIN-locked. As long as the device is not powered-off, one of its DMA controllers can be programmed over a DMA interface.

One form of defense is using an IOMMU, a memory management unit often found on PCs and laptops today. The OS can program the IOMMU to restrict what memory regions different DMA devices can access. Unfortunately, there are two problems with IOMMUs: (1) some mobile devices lack IOMMUs today, and (2) IOMMUs cannot authenticate DMA devices and are thus susceptible to spoofing attacks in which a malicious DMA device can impersonate another device. Therefore, IOMMUs must be programmed to deny access to a restricted memory range from *all* DMA devices.

Although some ARM platforms lack IOMMUs, most of today's ARM platforms are equipped with ARM TrustZone [2], which is ARM's hardware support for trusted computing. ARM TrustZone provides two virtual processors backed by hardware access control. The software stack can switch between the two states, referred to as the *secure world* and the *normal world*. The OS and all applications run in the normal world, whereas a small trusted kernel runs in the secure world. Untrusted code running in the normal world cannot access protected resources, such as memory pages and peripheral registers of the secure world. ARM TrustZone can protect against DMA attacks by giving the secure world control over the memory storing secrets, and denying normal world's DMA requests.

3.2 Out-of-Scope Threats

Data residing on a smartphone face many threats other than those described above.

Software Attacks. This class of attacks compromise the software on a mobile device and can thus access and steal any secrets, such as the owner's PIN number. Such attacks exploit a software vulnerability to install compromised code. Although an important class of attacks to consider, this paper focuses on attacks that do not rely on running compromised software, yet are possible when the device is in an attacker's hands.

Physical Side-channel Attacks. Side-channel attacks aim to obtain sensitive information by exploiting physical properties of the cryptographic implementation, such as timing information or power consumption. For example, different keys could affect the amount of processing time or power required to do the encryption differently, revealing something about the keys' structure. Although our system does address one class of side-channel attacks based on observing memory access patterns using bus monitoring, we do not address physical externally measurable side-channel attacks. Such attacks fall outside our threat model because they require a relatively high level of sophistication particularly when the attacker cannot run arbitrary code on the device.

Code-injection Attacks. This class of attacks inject or modify code running on the platform. One way to mount this attack is to use a compromised DMA-capable device. However, as described above, an ARM TrustZone (or an IOMMU) can protect a memory region against all DMA accesses including writes. Another possibility is to attach a logic analyzer to the memory or I/O bus and issue write commands that bypass any ARM TrustZone or IOMMU protections. We consulted a DRAM hardware expert about these attacks and learned that, albeit possible, these attacks are very challenging to mount because they are electrically unsound [8]. An attacker needs to use a higher drive strength than the bus to effectively "override" the bus protocol. Using a higher strength has the potential to damage the memory controller beyond repair. Also, bus protocols use dynamic time synchronization; the attacker must discover and synchronize the analyzer's timings to match those of the bus. In the expert's opinion, mounting such an attack costs several 100's of thousands of dollars, at a minimum.

JTAG Attacks. JTAG attacks are preventable because JTAG can be disabled. Some mobile vendors disable JTAG by depopulating the JTAG connector. Unfortunately, the JTAG interface can be re-enabled by soldering a JTAG cable back to the JTAG pad [38]. Other mobile vendors use a more secure form of JTAG-blocking based on hardware fuses [46]; such fuses can be disabled at device provisioning time. JTAG is permanently disabled once this fuse is burned. Additionally, manufacturers are also starting to build "authenticated JTAG", an interface that requires a reader to authenticate to enable JTAG [20].

Sophisticated Physical Attacks. It is hard to anticipate all types of physical attacks. One example of an advanced attack is using an electron microscope to examine the internals of the ARM SoC chip and carry attacks on the stored data [49]. Although possible, such an attack requires specialized equipment and can often take several months even when carried out by a skilled attacker.

4. On-SoC Storage

This section presents two alternatives for storing secrets on the SoC when the device is screen-locked.

4.1 Alternative #1: Internal RAM (iRAM)

Many smartphones and tablets are equipped with a small amount of internal SRAM on the SoC, called iRAM. By placing iRAM on the chip, different SoC subsystems (e.g., the DSP and various microcontrollers) have fast access to a little bit of memory to run their

Memory Preserved	iRAM	DRAM
OS Reboot (no power loss)	100%	96.4%
Device Reflash (power loss)	0%	97.5%
2 Second Reset (power loss)	0%	0.1%

Table 2. The iRAM (SRAM) and DRAM's data remanence rates on a commodity tablet.

firmware. For example, our Tegra 3 development board provides 256KB of iRAM.

However, like DRAM, SRAM storage also suffers from *data remanence* – the memory retains its contents for a period of time upon a power disconnect. Data remanence makes memory vulnerable to cold boot attacks [25]. In fact, SRAM decays over time more slowly than DRAM [10], making it even less suitable for storing secrets. We performed the following experiment to characterize iRAM's data remanence.

Methodology. We connected the Tegra 3 tablet to a debugging board and used its reset button to power off the tablet for short periods of time. For each experiment, we varied how long the power was cut from the board and measured the amount of data preserved in DRAM and iRAM, respectively. To do so, we created a process that allocates 1GB of memory (the tablet has 1GB of DRAM and 256KB of iRAM) and fills it with an 8-byte pattern written repeatedly.

We then performed three types of board resets that correspond to mounting different variants of cold boot attacks. First, we just performed a simple OS reboot. Such an attack is possible for *unlocked* mobile devices – an attacker can boot a malicious OS that dumps the memory. Second, we tapped the reset button (i.e., a short power disconnect), which corresponds to the reflashing the device. The final test is a longer power interrupt – we held the reset button pressed for two seconds. This corresponds to an attack in which the memory is yanked out of the device and quickly inserted into a malicious device which dumps all memory contents.

For each test, we used a kernel-level driver to enumerate all DRAM and iRAM before starting the test and after the test ended. We then grepped for the pattern, counted the number of occurrences, and used these results to compute data remanence as a ratio. Each test was per formed at room temperature, and was repeated five times. Table 2 shows the average results. Previous work has characterized DRAM and SRAM's memory remanence as a function of temperature [10, 25, 43].

Results. iRAM lost all its contents every time the device lost power no matter for how long. In contrast, at room temperature, DRAM still preserved a portion of its contents (0.1%) even when losing power for two seconds.

After private conversations with the ARM board's hardware manufacturer, we learned that the board's firmware zeroes out its iRAM upon boot up. This behavior is consistent with our results, and makes iRAM attractive for our security needs.

4.2 Alternative #2: Locked L2 Cache

Cortex-A9 ARM platforms are equipped with a PL310 cache controller [3] whose role is to manage a shared L2 cache. The PL310 offers the ability to lock a portion of the cache to prevent it from being evicted from the cache. ARM platforms have supported cache locking for a long time to improve computation performance as well as the predictability of computation latency. Being able to *pin* data into the cache can make small amount of computation faster and its duration predictable. This piece of functionality is especially attractive for embedded systems that need real-time guarantees. The PL310 controller mandates the ability to lock a cache *by way*, and optionally by *cache-line*. This is done via a Verilog option provided to the SoC vendors. Flushing the cache is done with a single instruction, regardless of the number of ways being flushed. We were able to enable cache locking by way on our NVidia Tegra 3 board. Unfortunately, we could not enable cache locking on the Nexus 4 because we do not have access to the boot firmware.

The locked L2 cache is useful for storing secrets on the SoC only if it *guarantees* that locked ways are not invalidated and remain present in the L2 cache until unlocked. While such a guarantee is not important for performance (the original goal of cache locking), it is crucial to our security needs. Unfortunately, the ARM spec is silent on whether the L2 cache locking implementation offers this guarantee. To validate our hardware behavior, we performed the following two experiments.

	iRAM	Locked L2 Cache
Cold Boot	Safe	Safe
Bus Monitoring	Safe	Safe
DMA Attacks	Safe (ARM TrustZone)	Safe

Table 3. Security analysis of different storage alternatives to DRAM.

Validating PL310's write-back behavior. First, we inspected the codebase of an ARM simulator built by ARM itself (ARM Fast Model [1]) and found that the PL310 does not write back locked entries. Our second experiment was done on our Tegra 3 board. We started by choosing an 8-byte random pattern that never appears in DRAM. We wrote the random pattern at the physical address that maps into a locked cache way, and used DMA reads to read the DRAM directly bypassing the cache.

Finding a way to "hijack" a DMA controller for our testing needs proved to be challenging for several reasons. First, few Linux drivers appeared to use DMA for their data transfers. Second, DMA controllers transfer data from memory to a device only. Reading the DMA-ed data from the device is sometimes impossible. For example, the NIC only allowed DMA-ing data out to a transmit buffer that cannot be DMA-ed back in. Incoming DMA transfers from the NIC were filled from a separate receive buffer. Third, certain devices expect the DMA-ed data to have a certain format; for example, the NIC expects transmitted data to have packet headers.

In the end, we used a driver that offers a high-speed serial channel. We discovered that the UART controller offers a *debugging* port that acts like a loopback interface – it returns all data written to it. We then modified the driver to DMA data to this debugging port and then read the serial port to output its contents. This experiment revealed that the hardware does not evict data from a locked cache way. However, we discovered that flushing the entire cache does unlock all locked ways. To ensure data safety, our system disables flushing of locked cache ways. We validated that the locked way's entries never appear in DRAM once this OS change is made.

4.3 Security Analysis of Storage Options

Table 3 summarizes our analysis of secure storage choices for protecting secrets against memory attacks.

Because the low-level firmware zeroes-out iRAM upon boot, storing secrets in iRAM protects them against cold boot attacks. Unfortunately, we do not know of any published data on how prevalent this behavior is across different types of mobile devices, and as a result we cannot generalize our finding beyond our Tegra 3 device. iRAM is also safe against bus monitoring attacks because secret data never leaves the SoC and thus does not traverse any externally exposed busses. Finally, iRAM is safe against DMA attacks as long as access to iRAM is protected inside the ARM TrustZone only, as described in Section 3.

Similar to iRAM, upon boot the low-level device firmware resets the PL310 L2 cache controller and zeros the L2 cache contents. As a result, sensitive data in the L2 cache is protected from cold boot attacks. Bus monitoring attacks are also ineffective because data does not travel on any exposed busses. On ARM platforms, DMA attacks are ineffective because DMA transfers data directly from DRAM, bypassing the L2 cache.

Because the low-level device firmware implements the zeroing logic for both iRAM and the L2 cache, one attack vector would be to replace this firmware. However, this attack is difficult to mount because the device verifies that the low-level firmware is signed with the manufacturer's key.

4.4 Trade-Offs

Looking at Table 3 one might ask: *What are the security trade-offs between iRAM and L2 cache locking because both appear to be safe to these attacks?* On one hand, any on-SoC storage is "safe" because placing sensitive on the SoC protects it from memory attacks. On the other hand, the hardware, the operating system, or even an application could accidentally copy data off the SoC to DRAM (even if not malicious). It is thus important to examine the security guarantees each on-SoC storage alternative offer.

The key distinction between cache locking and iRAM arises in how they protect against DMA attacks. On today's ARM SoCs, cache coherence for DMA transfers is handled in software. The operating system must explicitly flush the cache before DMA reads, and invalidate the cache before DMA writes. Because software manages the cache for DMA operations, hardware initiated DMA attacks cannot see the contents of the L2 cache, and therefore L2 cache-locking protects against DMA attacks. In contrast, iRAM is just like any other system memory (DRAM) with respect to DMA attacks. iRAM can only be protected from DMA attacks when software in the TrustZone takes explicit steps to protect it.

4.5 Implementation

iRAM. We implemented a simple memory allocator that manages the 192KB of iRAM (out of a total of 256KB) on our Tegra 3 tablet. In our experiments, we discovered that the first 64KB of iRAM appear to be used by our tablet's firmware; overwriting this region of iRAM crashes the tablet.

L2 cache locking. Locking a cache way is done by programming the PL310 controller with a special *enable way* command. This command has two effects. First, it ensures that all new data loaded in the cache will be loaded in the *enabled* way. Second, all data already loaded in the remaining *disabled* ways remains in fact available to reads and writes but no new allocations or evictions will take place. We can leverage this *disable way* command to first load sensitive data into one way, and then lock the way. In pseudocode, cache locking is done with the following four steps:

```
1. flush entire cache
2. enable 1-way
/* cache size is now 128KB */
3. write 0xFF in all sensitive data
/* warm the one-way with data */
4. enable last 7-ways
/* first way is now ``disabled'' */
/* cache size is now 7x128KB */
```

Once a way is locked, we return pointers to free pages mapped to this cache way. Once the entire way has been allocated, we lock an additional way if new requests for locked L2 cache memory are made. Section 8 will show the increasing performance overhead as

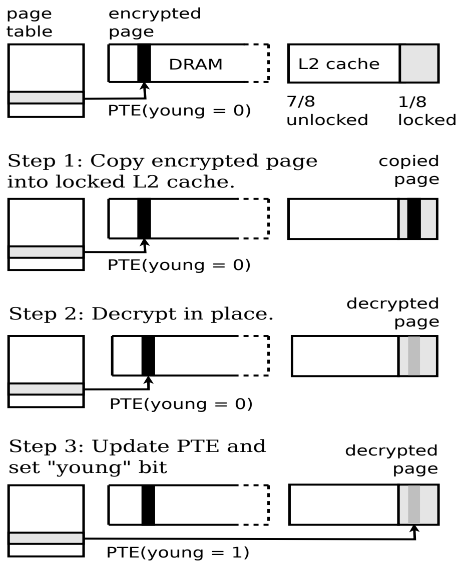


Figure 1. Decrypt on page-in.

additional ways are locked. To unlock a cache way, we erase the sensitive data and re-enable all unlocked ways:

```
1. write 0xFF in all sensitive data
/* all sensitive data erased */

2. enable (8-k)-ways
/* restore unlocked cache ways */
```

OS-level changes for handling L2 cache flushes. Flushing the L2 cache unlocks all locked ways. To ensure that data remains locked, we used a feature of the PL310 cache controller – supporting a mask that specifies which ways should be flushed. Each time a way is locked, our system sets this mask appropriately, and resets it when the way is unlocked. We changed all calls to the L2 cache flush routines in Linux to use this mask. As a result of these changes, the code handling L2 cache flushes in Linux grew from 428 to 676 lines of code.

5. Encrypted DRAM

This section describes how Sentry uses paged virtual memory to secure background applications. Sentry uses paging to manage data transfers between on-SoC memory and main memory (DRAM). Sentry ensures that pages stored in DRAM are always encrypted, and Sentry performs decryption on page-in to on-SoC memory, and encryption on page-out of on-SoC memory.

We clear a bit (called the *young* bit in the ARM architecture) of a Page Table Entry (PTE) to ensure we trap (page fault) whenever this page is accessed. Upon a page-in, the trap is fired, and we copy this page into a locked L2 cache way, and decrypt it in place. Once decryption is done, we modify the PTE to point to the decrypted page copy stored in the locked way and reset the "young" bit. Figure 1 illustrates the steps performed by Sentry during decryption on page-in.

If the locked L2 cache is full, Sentry needs to evict a page. For this, the same sequence is repeated in reverse. We encrypt in place, and then copy the page back, modify the PTE to point back to this location, and clear its young bit to ensure future traps.

While this approach supports running unmodified applications securely in the background, it has a couple of limitations. First, system performance is reduced as page faults need to be handled in software to enable careful control of the L2 cache contents. In addition, the limited capacity of the locked cache ways leads to an increase in the frequency of paging content in and out of the cache. Second, direct accesses to physical memory that do not trigger page faults, such as DMA by I/O devices, is not supported. We expect these limitations to be negligible because typical background jobs running on mobile devices (e.g., polling for email, periodically reading sensor data) have modest processing requirements.

6. On-SoC Encryption

To ensure protection from memory attacks, Sentry's implementation of encrypted DRAM cannot use generic encryption and decryption routines. Instead, the encryption and decryption routines used by Sentry must ensure that they do not leak any sensitive state to DRAM. This section presents AES_On_SoC, a secure implementation of the Advanced Encryption Standard (AES) [34] that prevents sensitive encryption state from leaking out to DRAM. We start with an in-depth analysis of AES to classify its state and determine what portion must be protected. We then describe the implementation steps taken to build AES_On_SoC.

6.1 An In-depth Analysis of AES's State

Brief AES Primer. The Advanced Encryption Standard (AES) [34] is a block cipher that operates on 128-bit blocks of plaintext and can use keys of 128, 192, or 256 bits. AES performs up to four different operations repeatedly, where each repetition is called a *round*. In each round, AES uses a specific, per-round encryption key, derived from the original encryption key. For performance reasons all round keys are precomputed and "cached" because they depend on the original encryption key only, and not on any intermediate state. While this optimization speeds up performance, it also increases the amount of AES sensitive state. This raises an interesting trade-off – a faster AES implementation requires more secure storage.

To precompute the round keys, AES uses two lookup tables, called the *S-box* and the *R-con*. While these lookup tables contain no secret state, the specific order in which AES looks up into these tables can leak sensitive information about AES.

To handle input data longer than 128 bits, AES breaks the input data into 128-bit input blocks and applies the single-block operation repeatedly. The ciphertexts corresponding to each block are then combined or chained using different modes of operation. AES supports many different modes (e.g., ECB, CTR, CBC). Sentry uses the CBC mode; CBC is also the default AES mode of operation on Android and Linux.

State Classification. Different types of state can be safe against some types of memory attacks but not against others. AES's state can be classified as follows:

1. Secret state. Secret state is state that, if leaked, will lead to a compromise. In AES, the input text, the encryption key, and most of the state derived from the encryption key is secret. Note that not all state derived from the encryption key is secret; for example, the ciphertext is not secret.

2. Public state. Public state is state that will not lead to a compromise if leaked. For AES, the public state is the ciphertext and any additional state that keeps track of the progress of the encryption. For example, the variable keeping track of the encryption round in AES is public.

3. Access-protected state. Although not secret, this state's access patterns are sensitive. For example, AES (and other symmetric encryption algorithms) use pre-computed lookup tables to speed up computation (e.g., the R-con and S-box tables). The lookup tables' contents are not secret, but monitoring what entries in these tables are accessed during encryption can reveal sensitive information about encryption keys.

	AES-128	AES-192	AES-256	Sensitivity
Input block	16	16	16	Secret
Key	16	24	32	Secret
Round Index	1	1	1	Public
Round Keys	320	368	416	Secret
2 Round Tables	2048	2048	2048	Access-protected
2 S-box	512	512	512	Access-protected
Rcon	40	40	40	Access-protected
Block Index	1	1	1	Public
CBC block/ivec	16	16	16	Public

Table 4. The breakdown of AES state in bytes.

How much state does AES use? Table 4 presents the state AES uses for encrypting with different key sizes. The input block holds the plaintext at the beginning of encryption and the ciphertext at the end as each round is performed on the same input block. We list its size as one block of 16 bytes (128 bits); however, the input can be arbitrarily long. Summing up the sizes of each piece of state leads to 2970 bytes of state for implementing encryption and decryption in AES-128.

What Portion of AES State is Sensitive? The last column of Table 4 classifies AES's state according to its sensitivity. The bulk of the state is access-protected and is due to the round tables. In fact, the round tables alone account for an order of magnitude more state than the rest of the state variables combined. In summary, the OpenSSL AES-128 implementation has 352 bytes of secret state, 2600 bytes of access-protected state, and 18 bytes of public state.

6.2 AES_On_SoC Implementation

Sentry implements the following two AES_On_SoC versions:

1. iRAM AES_On_SoC. This version stores all secret and access-protected stated in iRAM. This required few changes to the generic OpenSSL AES version that loads its state from DRAM.

2. Locked L2 AES_On_SoC. This version ensures that all the secret and access-protected state is allocated in a locked L2 cache way. On our platform, the size of one way is 128KB, which is plentiful to store all AES's sensitive state.

Handling context switches. Irrespective of where it is stored, sensitive state is loaded in the regular CPU registers at runtime to perform computation. If left unhandled, a context switch could evict all the sensitive data from the CPU registers and place it on the stack in DRAM. To avoid this situation, computation handling sensitive state is performed with interrupts raised. For this, we created two different macros called *onsoc_disable_irq()* and *onsoc_enable_irq()*. Our AES_On_SoC implementations encapsulate the computation handling secure state within these two macros. The first macro disables interrupts whereas the second macro zeroes all regular registers and re-enables interrupts. Our implementation raises interrupts for 160 microseconds on average on our Tegra 3 board.

Handling procedure calls. Sensitive state can also be put on the stack when passed as arguments to procedure calls. According to the ARM procedure call standard [4], the first four arguments to a procedure call are passed in registers, and the rest on the stack. The sensitive state is safe as long as is passed in the first four arguments to a procedure call. We inspected the AES_On_SoC implementation and checked that no procedure called used more than four arguments. This guarantees that all state is passed in registers and never put on the stack.

7. Prototypes

We developed two Sentry prototypes: one on an NVidia Tegra 3 ARM development board, and one on a Google Nexus 4 smartphone. Tegra 3 allows us the low-level access needed to enable cache-locking. This prototype fully implements Sentry including support for secure on-SoC encryption and background applications. In contrast, Nexus 4 does not support cache locking, and therefore cannot run background applications while locked. However, Nexus 4 enabled us to run experiments with widely used mobile apps, and to collect meaningful power measurements. Instead, our Tegra board is not as optimized for low energy consumption as a retail device, such as the Nexus 4.

Secure On Suspend. Our Nexus 4 prototype takes advantage of the fact that smartphones seek to suspend their state to DRAM frequently (this corresponds to the ACPI S3 state) to save energy. These suspend events occur after brief periods of user inactivity, or when the user turns off the phone by pressing the power button. Wake-up occurs (1) upon detecting user interaction, such as the user pressing the home or camera buttons, or (2) when some hardware event fires, such as an incoming phone call, or when some period timer fires.

Selective Encryption. On a device lock, a strawman design would encrypt all user-level state and most of the kernel state, except for kernel state needed for resuming the device, some low-level device drivers (e.g., cell telephony driver), and the code that handles decryption and device unlock. Despite its security appeal, such an approach incurs significant latency and energy cost.

Our experiments showed that encrypting 2GB of DRAM contents on a modern smartphone (Nexus 4) takes over a minute even when using all its four CPU cores and its crypto accelerator simultaneously. This also had a steep energy price: a single full-memory (2GB) encryption consumed over 70 Joules, completely depleting the battery after 410 suspend/resume cycles only. Hardware trends also do not look promising – while future smartphones will be equipped with faster CPU cores, their DRAM sizes will also be increasing, and energy consumption will continue to be a bottleneck. Given that a typical user consults her phone on average 150 times per day [5], to be practical Sentry must add little overhead in terms of time and energy.

Instead, Sentry only secures selected applications and OS subsystems identified as sensitive. On Android, users mark applications for encryption using an extension to the systems setting menu to chose the ones from a list of installed apps.

Upon device locking, Sentry walks the page tables of all processes marked sensitive and encrypts them. The Tegra prototype relies on AES_On_SoC for encryption. The Android prototype uses the default Linux Crypto API for encryption, and stores the encryption key in iRAM. In addition, the Android prototype marks encrypted processes as *un-schedulable* and places them in a special queue to prevent them from running in the background while the phone remains locked.

We also ported AES_On_SoC to the Linux Crypto API to ensure that the encryption does not release sensitive data outside the SoC. Any legacy software already designed to use this API automatically works with our system. We register our AES implementation with the API, providing it with a higher priority than the default AES implementation. Thus, if both the generic AES and our AES are loaded, the crypto system will favor ours.

On-demand Decryption. Sentry decrypts on demand using a lazy approach to minimize the impact of decryption to resume latency and energy consumption. We modified the page fault handler to enable on-demand page decryption. Encrypted pages have modified page table entries (PTE) to generate a trap next time the page is accessed. When the page fault is generated, Sentry decrypts the page in the context of the page fault handler. By using a lazy approach, Sentry saves energy and time in the case when users unlock their phones, engage in a just a few interactions, and re-lock their phones.

While most memory is decrypted on demand, Sentry handles DMA regions used by the GPU and I/O devices differently. These regions do not trigger page faults when accessed directly by their I/O devices using physical addresses. To handle these cases, Sentry eagerly decrypts DMA regions once the device is unlocked. These regions can vary in size, from 1MB for the Contacts application, to 3MB for Twitter, and 15MB for Google Maps.

A final case is handling memory pages shared between applications. If a memory page is shared with an application deemed non-sensitive, Sentry assumes that the contents of this memory page are not secret and skips encrypting it. However, if the page is shared only between sensitive applications, Sentry encrypts the page.

Securing Persistent State. Another concern is handling the persistent state of sensitive applications. Using file-system encryption only is not sufficient because the cryptography library could leave sensitive data in DRAM during execution. Instead, we incorporate AES_On_SoC in *dm-crypt*, a transparent block-level encryption module in Linux. At a high-level, *dm-crypt* makes three calls to an AES library, one to set the encryption and decryption keys, and two calls to encrypt and decrypt data. As described earlier, we incorporated AES_On_SoC into the kernel's Crypto API and modified dm-crypt to use AES_On_SoC instead of the default kernel AES library.

Securing Freed Pages. Another concern is handling memory pages that have been de-allocated (i.e. "freed") by a sensitive application because such pages could contain sensitive data. Although Linux has a kernel thread whose job is to zero-out these freed pages, there is no guarantee when this is done. This is not a risk for background applications because Sentry always encrypts these pages when paged out from the L2 cache even when freed. However, this is a potential risk for applications encrypted at screen-lock. This risk is eliminated by waiting for the kernel thread to zero-out all freed pages before locking the screen. We measured this overhead on our Nexus 4 device and found it negligible – the kernel thread can zero-out pages at a rate of 4.014 GB/s and an energy cost of 2.8 micro-Joules per MB.

Bootstrapping. Sentry uses two root keys for AES encryption, a volatile root key and a persistent one. The volatile key is used to encrypt all sensitive applications' memory pages. The volatile key is stored on the SoC and is generated upon each platform reboot with different values. The persistent key is used to encrypt all on-disk, persistent state (using dm-crypt). This key is derived from a combination of user input (a password entered at boot time) and a secret value stored in a secure hardware fuse on the device. Present mobile devices are already equipped with such a secure fuse storing a random, hard-to-guess number only readable by code running inside ARM TrustZone. We implemented code that reads the secure fuse and generates an AES persistent root key, although this code is not integrated into Sentry yet.

Minimum On-SoC Memory Requirements. The minimum amount of on-SoC memory required to implement Sentry is only two pages. Sentry requires at least one page for AES_On_SoC, and another for the code and data of the sensitive application being decrypted/encrypted. We verified that our Tegra 3 Sentry prototype works in this configuration, although the system becomes very slow because of very frequent page faults.

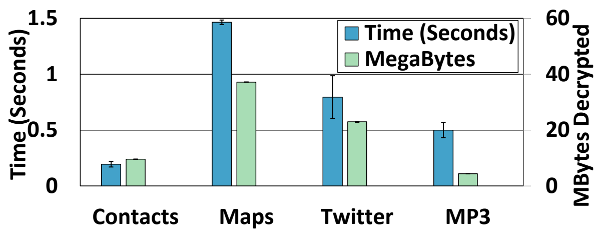


Figure 2. Performance overhead upon device unlock.

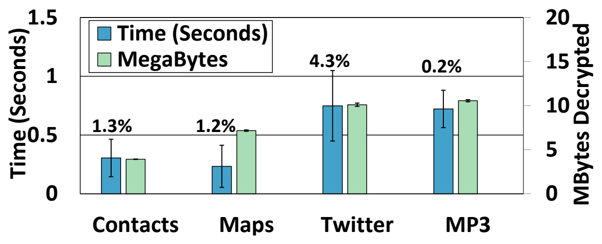


Figure 3. Performance overhead at runtime.

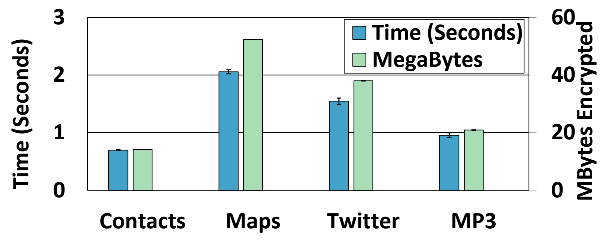


Figure 4. Performance overhead upon device lock.

8. Evaluation

This section evaluates the energy and performance overheads of Sentry in its various modes of protection.

8.1 Methodology

To evaluate Sentry, we use our two different hardware platforms: (1) an NVidia Tegra 3 ARM development board running Ubuntu Linux that allows us the low-level access needed to enable cache-locking, and (2) a Google Nexus 4 smartphone running a full-featured version of the Android OS. On the Tegra, we run a custom version of the Linux kernel v3.1.10, and on the Nexus the Linux kernel version is v3.4.0. The Tegra 3 contains a quad-core Cortex A9 CPU running at 1.2 GHz with 1GB of RAM, and the Nexus 4 contains a quad-core SnapdragonS4 CPU running at 1.5 GHz with 2 GB of RAM. All experiments are repeated at least ten times. All our graphs plot the average and the standard deviation, although sometimes the standard deviation is so low that it can barely be seen in the graph.

All mobile devices we are familiar with disable L2 cache locking in firmware. Because we only have access to the Tegra board's firmware, this is the sole platform where we can run our cache locking experiments. We only perform energy measurements on the

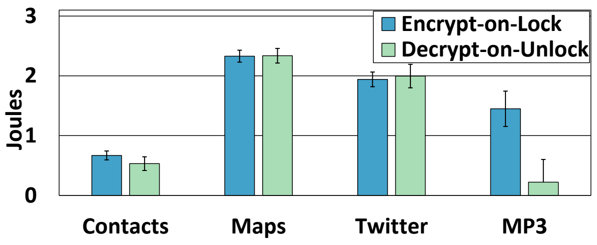


Figure 5. Energy overhead of encrypt-on-lock and decrypt-on-unlock.

Nexus 4 platform because our Tegra development board is not optimized for low energy consumption required by retail devices.

8.2 Performance and Energy Overheads

We use a series of application-level benchmarks to characterize the performance and energy overheads of Sentry. We then follow with microbenchmarks that show the raw AES energy and performance characteristics on our platforms.

Macrobenchmarks

We consider five different aspects of Sentry's overhead.

1. What is the performance overhead of Sentry on device lock and unlock? We run four Android applications, Contacts, Maps, Twitter, and ServeStream (an MP3 streaming app) both with and without Sentry on the Nexus 4. We measure their performance throughout their lifetime, from device unlock (where the memory pages of these applications start being decrypted), to running the applications, to device lock (where the memory pages of these applications must be all encrypted).

Device Unlock: Our first experiment measures the overhead added by Sentry of unlocking the Nexus 4 and performing a simple resume step of a single, sensitive application. This step decrypts only that portion of the application's memory pages needed to resume the application. Figure 2 shows Sentry's overhead in terms of both time and megabytes of data decrypted. We find that resuming sensitive applications has modest overhead, taking anywhere from 200 ms (Contacts) to a second and a half (Maps). This overhead is roughly proportional to the amount of data to be decrypted.

Sensitive Application Performance: We scripted each of these applications to perform a series of typical tasks. We run the script right after unlocking the screen and measure its duration; the scripts for the four applications take about 17 seconds for Twitter, 20 seconds for Maps, 23 seconds for Contacts, and 5 minutes for the MP3 app. With Sentry, the script takes longer to run because the applications are decrypting their memory pages on demand. Figure 3 presents Sentry's overhead in terms of time and megabytes of data decrypted. While these applications run, Sentry's overhead is small ranging between 0.2% and 4.3%.

Device Lock: Finally, all memory pages of the applications must be encrypted before the device is locked. Figure 4 shows this performance overhead. Again, the overhead is modest adding between 700 ms to 2 seconds per application. As before, the overhead is proportional to the amount of data needed to be encrypted.

Energy Profile: Figure 5 shows the energy overhead of device lock and unlock operations for protecting one sensitive application. These results do not include the energy consumed by an application at runtime; our benchmarking results later will show the energy costs of encryption/decryption at runtime. As Figure 5 shows,

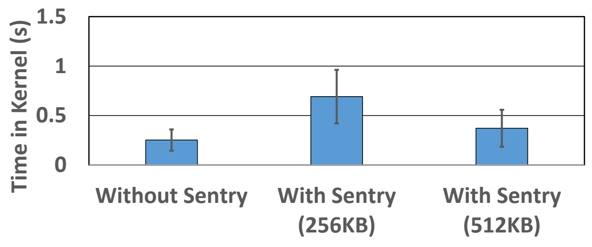


Figure 6. Performance of background computation for alpine.

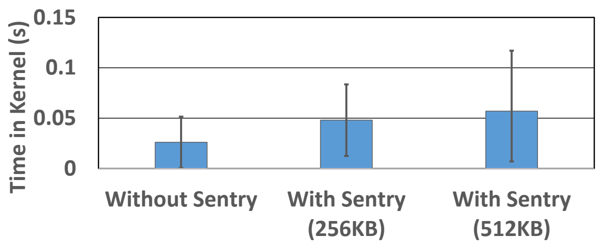


Figure 7. Performance of background computation for vlock.

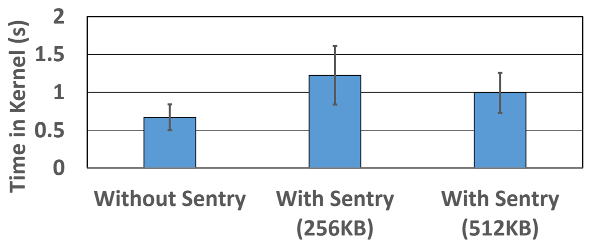


Figure 8. Performance of background computation for xmms2.

the overhead is minimal for all apps. Even for Google Maps, a relatively large application that needs to decrypt 38 MB of memory pages during device lock, and encrypt 48 MB, the energy overhead of device lock/unlock is modest consuming up to 2.3 Joules. We measure the unlock overhead conservatively because applications' memory pages are decrypted on demand rather than all at once as is the case with our experiment. Even so, Sentry will consume daily about 2% of a device's battery life to protect an application assuming the user locks and unlocks a phone 150 times a day.

2. What is the performance of Sentry's encryption and decryption for background computation?

On our Tegra 3 board, we ported three applications to Sentry: *alpine* (an e-mail reader based on *pine*), *vlock* (a text-based lock screen application), and *xmms2* (an MP3 player). These applications represent the types of actions users do when their smartphones are locked – they listen to music (xmms2), receive e-mail (alpine), and lock and unlock their devices' screens (vlock).

We run each application in the background for several seconds and measure the elapsed time spent inside of the kernel with and without Sentry. Figures 6, 7, and 8 show the results of our experiments. Sentry's overhead varies from a factor of 2.74 in the case of alpine when running with 256KB of locked L2 cache, to 48% in the case of xmms2 when running with 512KB of locked L2 cache. Although these overheads are high, we find Sentry's performance adequate because applications are generally not performance-critical while running in the background. Furthermore, these overheads are all relative to kernel time only, and not user time. Sentry's performance overhead is lower when considering an application's total time both in kernel and user-space. In all our experiments, we found that applications remain responsive when run in the background with Sentry.

3. What is Sentry's performance overhead for protecting file-system access with dm-crypt?

One of the most common ways to encrypt a file-system is using the *dm-crypt* Linux kernel module. Using Sentry to protect dm-crypt provides appealing security properties because it ensures that all persistent data in the system is secured against memory attacks.

We start by devising an experiment that isolates the performance overhead of Sentry when used on *dm-crypt*. *dm-crypt* performance is usually disk-bound, so we setup an in-memory disk partition of 450 MB and protect it with dm-crypt. We use three filebench [19] workloads: sequential reads, random reads, and random read/writes. All the dm-crypt benchmarks are run on the Tegra platform with cache-locking enabled.

For each run, the benchmark first creates a variety of files and then runs a sequence of I/O operations on these files, according to each workload. The initial file creation warms up the file system's buffer cache, and most of the I/O operations end up being serviced from the cache, without exercising the block-level AES encryption. To eliminate this effect, we also run the workloads using *direct I/O* to bypass the buffer cache.

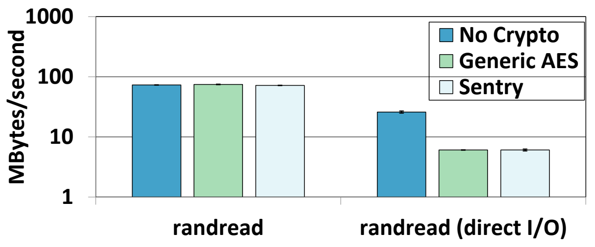
Figure 9 shows the throughput in megabytes per second of two workloads – running random reads (on the left) and random read/writes (on the right). Each graph has groups of three bars; the first bar corresponds to running with *dm-crypt* disabled, the second bar corresponds to running *dm-crypt* without Sentry, and the last bar corresponds to running *dm-crypt* with Sentry. The presence of the file system buffer cache "masks" some of the performance overhead of Sentry. Encryption adds no performance overhead for the randread benchmark. However, encryption cuts throughput by a factor of two for the randrw benchmark (the y-axis of Figure 9 is shown with a log-scale). When we eliminate the system buffer cache by using direct I/O, the impact of encryption on throughput is clearly visible.

4. What is the performance impact on the rest of the system when the L2 cache size is effectively reduced by cache locking?

Locking L2 cache ways can affect the performance of the rest of the system. We quantify this effect by running a CPU intensive task while locking different numbers of cache ways. Figure 10 shows, on the Tegra platform, the average duration of a Linux kernel compilation as a function of the number of locked cache ways. We use *"make -j 5"* as our build command, which runs up to five commands simulatenously during the build. As expected, the full compilation process takes longer as more ways are locked. However, the effect of locking a single way, which reduces the effective cache size by 128 KB (out of a total cache size of 1 MB) is not large. It takes 14.53 minutes to compile the Linux kernel with one locked way versus 14.41 minutes with no locked ways, an increase of 7.2 seconds (less than 1%).

Microbenchmarks

These microbenchmarks are useful in understanding sources of overhead in the application macrobenchmarks. On Tegra, we use three AES implementations: unsafe AES (unmodified OpenSSL AES), cache-locked AES, and iRAM AES. On the Nexus 4, we use three versions of AES: unsafe AES (unmodified OpenSSL AES)



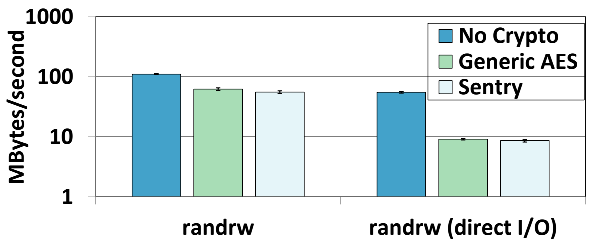


Figure 9. Performance of dm-crypt for random reads (left) and random read/writes (right).

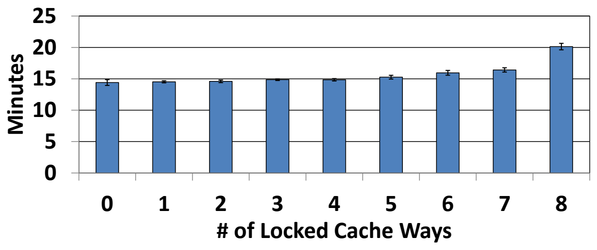


Figure 10. Performance of Linux kernel compile. Compiling Linux gets gradually slower as more cache ways are locked.

both at running in user-level and in the kernel, and the hardware accelerator AES.

Figure 11 shows the performance of AES for all the different AES variants on both platforms, with the Nexus 4 on the left and Tegra 3 on the right. At first, we were surprised to discover that AES is slower when running on the crypto hardware accelerator than on the CPU. However, upon investigation, this discrepancy is due to two factors. First, we are using 4KB blocks of data (the page size) to encrypt/decrypt. Crypto accelerators exhibit higher performance when encrypting large amounts of data. Second, our experiment is done upon phone lock and we suspect that the crypto accelerator is down-scaling its frequency. We repeated this experiment with the phone fully awake and the crypto accelerator is 4$\times$ faster. However, we were unable to prevent the crypto hardware from being down-scaled in our experiments. Without downscaling, the performance of our crypto accelerator is similar to previously published results [28].

Figure 11 also shows that Nexus 4 is much faster than our Tegra development board. Finally, and more importantly, on Tegra using AES_On_SoC adds negligible overhead (less than 1%).

Figure 12 shows the full-system energy overhead of CPU-based encryption versus hardware-accelerated encryption on Nexus 4. These results are driven by how long encryption takes using each method. Hardware-accelerated encryption is much less energy-efficient than the CPU due to the lower throughput of the AES hardware accelerator when encrypting 4 KB memory pages.

9. Related Work

9.1 On Chip AES Schemes

A few previous projects have investigated the idea of storing AES keys in CPU registers. In addition of being AES-for-x86-specific, most of these previous solutions fail to guard access-protected state and thus are subject to bus monitoring attacks. Furthermore, to us, it is unclear how to extend these solutions to safeguard the voluminous access-protected state.

In AESSE [32], Müller et al. implement AES for x86 processors by leveraging the SSE functionality of Pentium III and later processors to both store encryption and decryption keys and to perform the AES computation. Their initial implementation, based on the FIPS documentation [34], performed all the calculations sequentially on the input data, producing a very slow implementation. Their final implementation makes use of the table lookup optimization, improving performance from a 100$\times$ slowdown to a 6$\times$ slowdown over regular AES. In this implementation, all of the AES lookup tables (i.e., access-protected state) are stored in DRAM.

Their later work, TRESOR [33] expands on the AESSE functionality to support 64-bit CPUs with support for Intel's encryption instructions (AES-NI). This version is resilient to memory attacks because all AES's state is stored in AES-NI in the CPU. However, such an approach is not extensible beyond protecting AES implementations only.

Simmons [41] disables model-specific registers used for debugging and instead re-purposes them to store a single 128-bit AES key. This key is used to decrypt other keys stored encrypted in DRAM. This improves the performance to only a 2$\times$ slowdown over regular AES. The implementation's description specifies that no temporary data is stored in DRAM, although the paper offers no clear explanation on how large round tables are being stored in model-specific registers.

9.2 Other Forms of Defense

Encrypted RAM. Encrypted RAM schemes aim at storing data encrypted in DRAM. There are two main differences between our approach and encrypted RAM: (1) encrypted RAM's performance overhead is typically high and (2) it is unclear whether encrypted DRAM can protect against DMA attacks. In particular DMA transfers should be automatically decrypted otherwise there is little benefit of using DMA for fast memory access.

Cryptkeeper [35] expands the traditional memory hierarchy by proposing to encrypt a large portion of DRAM. Their implementation stores DRAM's encryption keys in the clear portion of RAM.

Another project [12] proposed the idea of encrypting the memory for processes when evicted from the cache. Like Sentry, they use a form of cache locking to ensure that sensitive data leaving the SoC is encrypted. However, this previous work differs from Sentry in four aspects. First, it only describes a preliminary design with no implementation. Second, the evaluation is all done in simulation on top of a Pentium-based architecture (whereas their design is aimed at ARM SoC) while artificially injecting hypothetical costs, such as the number of cycles encryption might take. Third, it is aimed at embedded systems and does not meet the needs of smartphones today. For example, there is no notion of "pin lock", no background processes, no discussion of DMA capabilities. Instead, a different set of assumptions are made, such as whether MMU is present or what types of crypto engines could be found on embedded systems. Finally, the cache-locking mechanism discussed is hypothet-

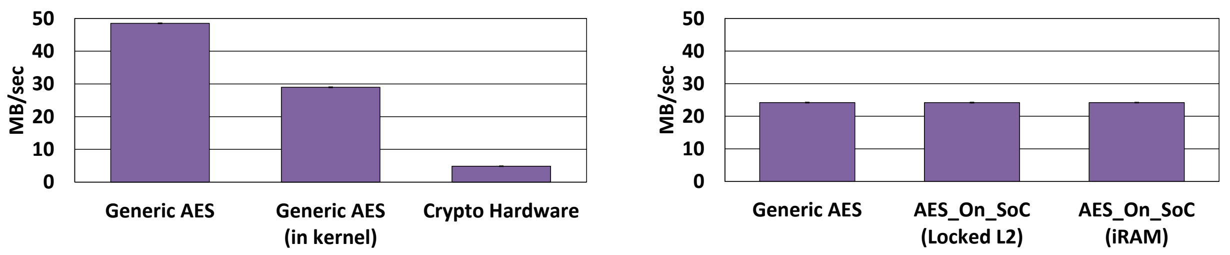


Figure 11. AES performance. Comparison of OpenSSL AES, the kernel CryptoAPI AES, and hardware-accelerated AES on Nexus (left), and OpenSSL AES, cache-locked Sentry, and iRAM Sentry on Tegra.

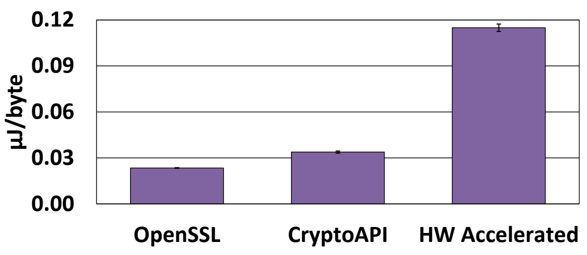


Figure 12. Energy overhead. Comparison of OpenSSL AES, the kernel Crypto API AES, the hardware-accelerated AES on Nexus.

ical, and thus it is unclear how the cache locks its contents (e.g., by way or by line).

Other methods have been proposed that store encryption keys in a secret place. ZIA [14] and Transient Authentication [15] use user-carried, small hardware tokens, and CleanOS [48] uses cloud servers for storing encryption keys. Unlike Sentry, such schemes rely on additional external infrastructure in the form of an Internet connection [48] or short-range wireless interfaces [14, 15]. Transient Authentication also does not support running background application securely while the phone is locked. Another difference lies in the type of abstractions used. CleanOS provides APIs that let applications define their sensitive data that needs protection. Instead, Sentry reduces the programmer's burden by encrypting all sensitive applications' state.

Auditing file-systems. Keypad [23] keeps logs of file-system accesses. While such an approach does not explicitly defend against memory attacks, it enables the victim to determine what portion of their data has been compromised when the mobile is lost.

9.3 Complementary Threat Models

A large body of work aims at protecting the integrity and confidentiality of trusted applications' code and data [13, 16, 17, 22, 26, 29, 37, 39, 40, 42, 44, 45, 47, 52]. These related works defend against a compromised OS (e.g., malware) or a compromised privileged VM, whereas Sentry relies on the OS to defend against DRAM attacks.

10. Discussion

ARM TrustZone: Today's ARM platforms offer TrustZone [2], a set of hardware extensions for code and data isolation. TrustZone can be used to isolate a set of secure services from the general purpose operating system. With TrustZone, the processor can execute instructions in one of two processor modes, referred to as the *normal world*, where untrusted code executes, and the *secure world*, where secure services run. One CPU instruction allows the software stack to switch back and forth between these two worlds. All memory allocated to the secure world remains inaccessible to the normal world. This offers a form of protection where malware in the normal world cannot infect the secure world. ARM TrustZone does not encrypt secure world memory.

Sentry uses TrustZone to enable cache-locking. This requires configuring certain co-processor registers which are only accessible from the secure world of TrustZone. We have access to TrustZone only on our Tegra 3 device, but not on the Nexus 4 (due to locked firmware). This is why we cannot test cache locking behavior on the Nexus 4.

Unfortunately, ARM TrustZone cannot help prevent cold boot or bus monitoring attacks. Removing RAM modules (one form of cold boot attack) and bus monitoring make the entire contents of RAM vulnerable, including any RAM allocated to the TrustZone. Reflashing the device firmware (another form of cold boot attack) also can bypass TrustZone's protection of secure world memory.

Running 3rd-party applications inside TrustZone makes them immune to DMA attacks. However, this approach brings significant challenges: running an OS/browser/DalvikVM in the TrustZone [39] and porting each app to it. The TrustZone's software stack could become very large, increasing the TCB of any other security services that rely on TrustZone.

Intel Software Guard Extensions (SGX): Intel has recently announced SGX [6, 27], a new set of CPU instructions for code and data isolation. Unlike TrustZone, the memory allocated to an "enclave" is encrypted in hardware, which renders cold boot and bus monitoring attacks ineffective. However, SGX brings in a set of additional challenges. First, OS code cannot run inside of SGX (SGX runs ring 3 code only). Second, based on the current documentation, SGX offers no form of protection against rollback attacks (i.e., no secure counters). Third, SGX technology appears to be aimed at the server market targetting cloud computing datacenters.

Architecture suggestions: This work demonstrates the feasibility and security benefits of locking data on the SoC. We believe that modern CPUs could offer a small amount of memory on the SoC together with a pin-on-SoC abstraction. Operating systems can make use of this abstraction to store cryptographic keys used to bootstrap additional system security, such as encrypted main memory. Such keys will be much better protected from memory attacks. In Sentry, we went to great lengths to use ARM-based mechanisms designed for computation performance and predictability (e.g., cache locking and internal RAM), and applied them to our security needs. However, Sentry's design would be greatly simplified in the presence of a simple abstraction that would guarantee a small amount of memory pinned on the SoC. This memory should be inaccessible to DMA controllers to further prevent DMA-based attacks.

To increase the security of on-SoC storage, we also recommend that low-level firmware should always erase it (i.e., zero it) upon device boot up. Additionally, this low-level firmware should not be modifiable to guarantee an attacker cannot simply disable it. Although this step would increase the time required for the device to boot up, the increase should be modest because zero-ing out a small amount of memory should be relatively fast. We believe this overhead is justified given the additional security gains.

11. Conclusions

This paper shows that today's smartphones and tablets have several performance features that can be retrofitted to defend against memory attacks. We implement Sentry, a system that leverages these features to guarantee that the sensitive state of a program is never stored in DRAM, but always stored on the ARM SoC. Our main observation is that sensitive state only needs to be encrypted when the device is screen-locked. However, to run computation in the background when the screen is locked, Sentry decrypts and encrypts the memory pages of sensitive applications as they are paged in and out. Sentry uses a carefully engineered AES crypto library (called AES_On_SoC) that never stores any sensitive state in DRAM. This combination of mechanisms ensures Sentry never leaks any sensitive state to DRAM when the device is screen-locked.

Acknowledgments

We would like to thank our shepherd, Gernot Heiser, and the anonymous reviewers for their thorough feedback. We would also like to thank Andrew Baumann and Jay Lorch for their feedback on earlier drafts of this paper.

References

[1] ARM. Fast models. http://www.arm.com/products/tools/models/fast-models/index.php. Accessed: 2014-12-10.

[2] ARM. ARM security technology – building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2005–2009.

[3] ARM. PL310 cache controller reference manual, 2007. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0246a/DDI0246A_l2cc_pl310_r0p0_trm.pdf.

[4] ARM. Procedure Call Standard for the ARM Architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf, 2012.

[5] T. T. Athonen and A. Moore. Commununities dominate brands. http://communities-dominate.blogs.com/brands/2013/03/. Accessed: 2014-12-10.

[6] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proc. of the 11th Symposium on Operating System Design and Implementation (OSDI)*, Broomfield, CO, 2014.

[7] M. Becher, M. Dornseif, and C. N. Klein. Firewire - all your memory are belong to us. In *Proc. of CanSecWest Applied Security Conference*, 2005.

[8] R. Bittner. Personal Communication, April 2014.

[9] A. Boileau. Hit by a bus: Physical access attacks with firewire. In *Proc. of 4th Annual Ruxcon Conference*, 2006.

[10] C. Cakir, M. Bhargava, and K. Mai. 6T SRAM and 3T DRAM data retention and remanence characterization in 65nm bulk CMOS. In *Custom Integrated Circuits Conference (CICC)*, 2012.

[11] E. M. Chan, J. C. Carlyle, F. M. David, R. Farivar, and R. H. Campbell. Bootjacker: compromising computers using forced restarts. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*.

[12] X. Chen, R. P. Dick, and A. Choudhary. Operating system controlled processor-memory bus encryption. In *Proceedings of the conference on Design, automation and test in Europe*, 2008.

[13] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. of 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, 2008.

[14] M. D. Corner and B. D. Noble. Zero-interaction authentication. In *Proc. of the 8th Annual International conference on Mobile computing and networking (Mobicom)*, 2002.

[15] M. D. Corner and B. D. Noble. Protecting applications with transient authentication. In *Proc. of the 1st International Conference on Mobile systems, applications and services (MobiSys)*, 2003.

[16] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proc. of 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[17] O. H. A. Dunn, S. Kim, M. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. In *Proc. of 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[18] EPN Solutions. Analysis tools for DDR1, DDR2, DDR3, embedded DDR and fully buffered DIMM modules. http://www.epnsolutions.net/ddr.html. Accessed: 2014-12-10.

[19] Filebench. Filebench: File system benchmark. http://sourceforge.net/projects/filebench/. Accessed: 2014-12-10.

[20] Freescale Semiconductor. Configuring secure JTAG for the i.MX 6 series family of applications processors. http://cache.freescale.com/files/32bit/doc/eng_bulletin/AN4686.pdf, 2013.

[21] FuturePlus System. DDR2 800 bus analysis probe. http://www.futureplus.com/download/datasheet/fs2334_ds.pdf, 2006.

[22] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proc. of 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, 2003.

[23] R. Geambaşu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An Auditing File System for Theft-prone Devices. In *Proc. of the European Conference on Computer Systems (EuroSys)*, 2011.

[24] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin. Reconfigurable hardware for high-security/high-performance embedded systems: The SAFES perspective. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(2): 144–155, 2008.

[25] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. of the 17th USENIX Security Symposium*, 2008.

[26] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. L. orzynski, F. Mehnert, and M. Peter. The Nizza secure-system architecture. In *Proc. of 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 2005.

[27] Intel. Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/329298-001.pdf, 2013.

[28] V. Keränen. Cryptographic algorithm benchmarking in mobile devices. Technical Report Master's Thesis, University of Oulu, 2013.

[29] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[30] Microsoft. BitLocker Drive Encryption. http://windows.microsoft.com/en-us/windows7/products/features/bitlocker.

[31] T. Müller and M. Spreitzenbarth. FROST - forensic recovery of scrambled telephones. In *Proc. of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2013.

[32] T. Müller, A. Dewald, and F. C. Freiling. AESSE: a cold-boot resistant implementation of AES. In *Proc. of the 3rd European Workshop on System Security (EUROSEC)*, 2010.

[33] T. Müller, A. Dewald, and F. Freiling. TRESOR runs encryption securely outside RAM. In *Proc. of the 20th USENIX Security Symposium*, 2011.

[34] NIST. Pub. 197 – advanced encryption standard (AES). `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, 2001.

[35] P. A. Peterson. Cryptkeeper: Improving security with encrypted RAM. In *Proc. of IEEE International Conference on Technologies for Homeland Security*, 2010.

[36] D. R. Piegdon. Hacking in physically addressable memory - a proof of concept. Presentation to the Seminar of Advanced Exploitation Techniques, 2006.

[37] H. Raj, D. Robinson, T. Tariq, P. England, S. Saroiu, and A. Wolman. Credo: Trusted computing for guest VMs with a commodity hypervisor. Technical Report MSR-TR-2011-130, 2011.

[38] Riff Box. `http://www.riffbox.org/`, 2014. Accessed: 2014-12-10.

[39] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *Proc. of 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[40] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus Authorization Logic (NAL): Design Rationale and Applications. *ACM Transactions on Information and System Security*, 14(1), 2011.

[41] P. Simmons. Security through amnesia: A software-based solution to the cold boot attack on disk encryption. In *Proc. of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.

[42] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *Proc. of 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[43] S. Skorobogatov. Low temperature data remanence in static RAM. Technical Report UCAM-CL-TR-536, University of Cambridge, Computer Laboratory, 2002.

[44] J. Sorber, M. Shin, R. Peterson, and D. Kotz. Plug-n-Trust: Practical trusted sensing for mHealth. In *Proc. of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[45] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proc. of European Conference on Computer Systems (Eurosys)*, 2010.

[46] STMicroelectronics. STM32F205/215, STM32F207/217 Flash programming manual. `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/programming_manual/CD00233952.pdf`, 2013.

[47] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operati ng systems configurable. In *Proc. of the 7th Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[48] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambaşu, and N. Sarda. CleanOS: Limiting mobile data exposure with idle eviction. In *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[49] C. Tarnovsky. Attacking hardware: Unsecuring [once] secure devices. Black Hat USA Training Session, 2009.

[50] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attaks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–31, 2010.

[51] TrueCrypt. Truecrypt – free open source disc encryption software. `http://www.truecrypt.org/`. Accessed: 2014-04-01; Product and website retired on: 2014-05-28.

[52] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. of 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.