SobTrA: A Software-based Trust Anchor for ARM Cortex Application Processors

Julian Horsch, Sascha Wessel, Frederic Stumpf, and Claudia Eckert
Fraunhofer AISEC
Garching near Munich, Germany
{firstname.lastname}@aisec.fraunhofer.de

ABSTRACT

In this paper, we present SobTrA, a Software-based Trust Anchor for ARM Cortex-A processors to protect systems against software-based attacks. SobTrA enables the implementation of a software-based secure boot controlled by a third party independent from the manufacturer. Compared to hardware-based trust anchors, our concept provides some other advantages like being updateable and also usable on legacy hardware. The presented software-based trust anchor involves a trusted third party device, the verifier, locally connected to the untrusted device, e.g., via the microSD card slot of a smartphone. The verifier is verifying the integrity of the untrusted device by making sure that a piece of code is executed untampered on it using a timing-based approach. This code can then act as an anchor for a chain of trust similar to a hardware-based secure boot. Tests on our prototype showed that tampered and untampered execution of SobTrA can be clearly and reliably distinguished.

Categories and Subject Descriptors: D.4.6 [Operating Systems]: Security and Protection

Keywords: Software-based Trust Anchor; Self-checksumming Code; Smartphone; Mobile Security; ARM Architecture; Secure Boot

1. INTRODUCTION

A widely used technique to improve embedded devices security and especially smartphone security is a *secure boot process* [1], which can be used to enforce that only signed code can be executed on the smartphone. Core of a *hardware-based secure boot* is the software stored in the Read-Only Memory (ROM) of the device. This first stage of the boot process, called *root of trust* or *trust anchor*, is the first code to be executed at system start. The main task of this trust anchor is to *measure* the next stage, i.e., to verify its signature, and to load it if the verification was successful, building the basis for a *chain of trust* controlled by the manufacturer. The hardware-based secure boot approach has some downsides. First, it can only be utilized by the authority that has control over the ROM of the device. Second, the secure boot may not be added to legacy devices. Finally, if the trust anchor code has a vulnerability or the key is compromised, there is no way to restore the security properties on the affected devices.

To overcome the aforementioned disadvantages, we propose the concept of a Software-based Trust Anchor (SobTrA) as the basis for a *software-based secure boot*. The concept is developed for the ARM architecture. SobTrA provides the guarantee that a piece of code runs untampered on the booting platform. With that, the software-based secure boot can establish a chain of trust on the *initially* untrusted platform. In contrast to the hardware-based concept, the guarantee is not provided to the manufacturer but to a trusted device, the *verifier*, which is connected to the untrusted platform. The verifier could, for example, be realized as microSD card issued by a trusted third party (e.g., a company) and connected via Secure Digital Input Output (SDIO) or Serial Peripheral Interface (SPI) to the untrusted platform. Figure 1 shows our concept for a software-based secure boot process. The software-based secure boot can conceptually be initialized at any point during runtime of the system. The basic idea is to let the verifier measure the time the untrusted platform needs to perform a self-checksumming calculation initialized by some challenge. If the verifier receives the correct result in a timeframe which does not exceed some previously determined upper bound, the untrusted platform is verified successfully. Because of the strong hardware dependency, existing primitives for *software-based externally verifiable untampered execution* cannot be simply used to realize SobTrA on ARM.

This paper is organized as follows. In Section 2, we describe our attacker model and assumptions. In Section 3, we give an overview of existing research. Section 4 introduces the basic concept and protocol of SobTrA. The ARM

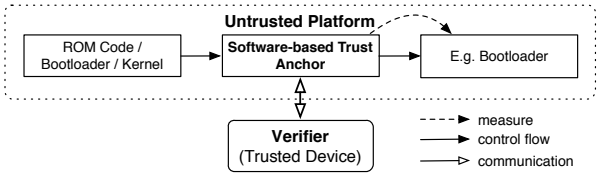

Figure 1: Software-based secure boot process

Cortex-A8 specific design and implementation of the SobTrA checksum function is presented in Section 5. Section 6 discusses the secure execution environment to be established in SobTrA. Finally, we present the experimental results on our prototype in Section 7 and conclude in Section 8.

2. ATTACKER MODEL & ASSUMPTIONS

The main goal of SobTrA is to prevent remote attacks, e.g., via the internet. Therefore, in our attacker model, the attacker has complete control over the software on the untrusted platform, but is not allowed to do hardware modifications or use hardware-based debug facilities (JTAG). Furthermore, DMA-based attacks are out-of-scope for this paper since they require platform specific countermeasures. Our assumptions for the execution of SobTrA are:

- The exact hardware configuration of the untrusted platform must be known to the verifier, including CPU type (e.g. ARM Cortex-A8) and clock speed.
- The verifier can be sure that the messages it receives are coming from the untrusted platform, excluding Man-In-The-Middle (MITM) attacks.
- No proxy or relay attack is possible. These are attacks where the untrusted platform uses the help of another device to fulfill the challenge from the verifier.

Especially in the context of a smartphone with a fixed, low-latency connection to the verifier and not allowing hardware modifications, these assumptions can be considered quite reasonable.

3. RELATED WORK

The first approach for externally-verifiable execution was proposed by Kennell and Jamieson in 2003 [9] for an x86 architecture platform. Their primitive, called *Genuinity*, is a *self-checksumming* function that additionally to its own memory layout includes performance counters, into the checksum. Unfortunately, the approach suffers from some serious problems as Shankar et al. describe in [17] (*substitution attacks*, see Section 4.2.1).

The first concept targeting embedded devices was SWATT, proposed in 2004 by Seshadri et al. [16] for an Atmel 8-bit microcontroller. SWATT tries to reduce the time needed for one iteration of the underlying checksum function main loop to a minimum, while still including some fast accessible CPU state inputs like the *program counter* and *status register*. This enables the verifier to measure even very small overheads introduced by attacker's code trying to circumvent the self-checksumming property. Another crucial advantage is that the execution time is only dependent on the number of iterations and not on the challenge itself. Therefore, it is *not* necessary to measure the "correct" time for a response from the untrusted platform for every challenge-response pair on a trusted device incorporating the target hardware. Later publications by Seshadri et al. refine the approach and adapt it to other architectures, namely x86 with *Pioneer* [15, 13] and TI MSP430 with *ICE* [14].

Newer publications introduce other concepts to implement checksum functions with similar properties. Either by generating the checksum function itself as the challenge [11] or by introducing a memory bottleneck during the checksum calculation to slow down the attacker's code [5, 7]. Downside

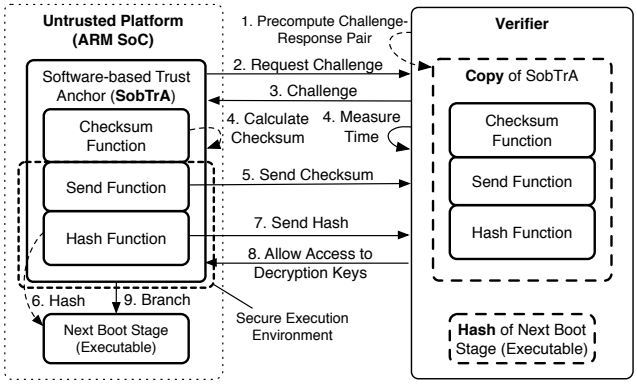

Figure 2: Basic concept of SobTrA and the software-based secure boot process

of the first approach is that it reintroduces a higher variance for the checksum function execution time. Downsides of the latter approach are that it takes more time to execute and is harder to be used at system runtime, requiring massive memory swapping.

SobTrA is based on the concepts by Seshadri et al. [16, 15, 14, 13] but presents a new secure boot use case and, most importantly, is designed and implemented on a different processor architecture, ARMv7-A, resulting in substantial changes to the underlying algorithm design and implementation.

4. SOBTRA CONCEPT

The typical hardware architecture for the SobTrA secure boot concept consists of a ARM System on a Chip (SoC) device as the *untrusted platform*, e.g., a smartphone, and a low performance device of arbitrary architecture as the *verifier*. Both are interconnected via a local, low-latency physical interface, such as SPI or SDIO. The core of the proposed software-based secure boot, as shown in Figure 1, is the software-based trust anchor SobTrA. An overview of the SobTrA functionality is depicted in Figure 2.

4.1 Basic Structure and Protocol

SobTrA on the untrusted platform consists of three main parts. First, there is the *checksum function*, which is the core of the software primitive. In the first communication step, SobTrA requests a challenge (2.). The verifier either pre-computes (1.) challenge-response pairs to accelerate the protocol or computes one on demand directly before sending the challenge back to the untrusted platform (3.). SobTrA uses the challenge to initialize the checksum function which then calculates a checksum (4.) over itself and the other two main parts of SobTrA while the verifier measures the time. Afterwards, SobTrA initializes a *secure execution environment* for the rest of the trust anchor to run in, preventing an attacker from gaining control via exceptions (see Section 6). The second part of SobTrA is the *send function* which is responsible for sending the result checksum back to the verifier (5.). The last part inside the checksummed region is the *hash function*, which computes (6.) a hash digest of the next stage in the boot process. The following steps depend on the application scenario. In our secure boot, the untrusted platform sends the hash digest to the verifier (7.) who checks if the hash is correct and only allows access to decryption keys/functionality (8.) if the verification is suc-

cessful. The keys could, for example, protect the root file system and therefore prevent a further boot. All parts of SobTrA must be self-contained, i.e., they must *not* call code outside the checksummed region and must not cause exceptions.

The verifier contains an exact copy of the SobTrA binary image. It is therefore able to calculate the correct checksum to validate the checksum received from the untrusted platform. Besides the challenge, the verifier also sends the number of loop iterations for the checksum function which determines the maximum time the untrusted platform is allowed to take for returning the correct checksum. This maximum time must be pre-measured *once* on a trusted device identical to the untrusted platform and is valid for *all* challenges with the same iteration number. After receiving the checksum result from the untrusted platform, the verifier checks if the time taken is below the pre-measured threshold and whether the checksum is correct. If both conditions are met, the secure boot can succeed.

4.2 Checksum Function Design

The checksum function must provide the following *basic property*: A *tampered* checksum function either yields a *wrong checksum* or causes a *measurable* execution time *overhead*. Figure 3 shows the basic structure of the checksum function in the context of the other SobTrA parts. The first step towards the basic property is to make the checksum function *self-checksumming*, incorporating itself and all other parts of SobTrA into the checksum. This forces an attacker, trying to modify a part of SobTrA, to manipulate accesses to these memory regions to avoid yielding a wrong checksum. This increases computation time and therefore generates a measurable overhead. The function mainly consists of one main loop, in which a word from memory is read, transformed and included into the checksum together with CPU state input. An attacker trying to circumvent the self-checksumming property always introduces overhead inside this main loop. Therefore, even very small overheads can be made measurable by simply increasing the number of loop iterations. The epilogue code is mainly responsible for initializing the secure execution environment for the rest of the trust anchor (discussed in Section 6).

4.2.1 Common Attack Types

There are two basic types of attacks against such a checksum function: Substitution attacks and memory copy attacks. In a *substitution attack*, the attacker replaces parts of the trust anchor code, for which he then tries to circumvent the self-checksumming property specifically. Figure 4 visualizes an example. In a *memory copy attack*, the attacker modifies the code but still keeps an untampered version somewhere in memory. The attacker's tampered code then computes the checksum by redirecting all memory reads

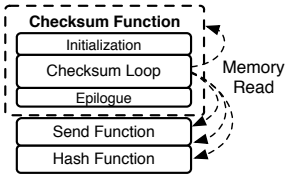

Figure 3: Checksum function structural overview

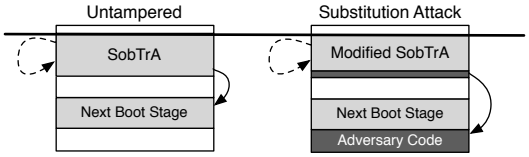

Figure 4: Substitution attack example: The final branch is substituted to redirect the flow to adversary code.

performed in the context of the self-checksumming functionality to the untampered image. Figure 5 illustrates the two basic forms of memory copy attacks.

4.2.2 Desired Properties

Seshadri et al. [16, 15] postulated some properties which should be fulfilled by a checksum function to provide the aforementioned basic property of either producing a wrong checksum or causing a measurable overhead when tampered. These are the basis for the SobTrA checksum function properties explained in the following: (1.) The memory which is getting checksummed should be *traversed in a pseudo-random manner* to prevent substitution attacks. (2.) The checksum function should include *CPU state inputs*, e.g., Program Counter (PC) and Status Register (SR), into the checksum, mainly to prevent memory copy attacks. (3.) The checksum function should be as *strongly-ordered and non-parallelizable* as possible to prevent addition of malicious instructions without overhead. (4.) Iterative parts, i.e., the main loop, of the checksum function should be *as small as possible* to allow the measurement of even very small attacker overheads and to reduce complexity for the code optimization. (5.) The checksum function implementation should be *time optimal*. The optimality could be made sure in the future using tools like Denali [8]. (6.) The checksum function execution time should have a *low variance*, i.e., only depend on the number of loop iterations to be able to do a verification for random challenges. (7.) To prevent precomputation of the checksum, the checksum function should be *initialized by a random challenge*. (8.) To slow down code added by an adversary (register spilling), the checksum function should *use all available CPU registers*.

5. CORTEX-A8 CHECKSUM FUNCTION

A checksum function implementation *must* be designed very specifically for a processor architecture to provide the introduced properties. SobTrA is designed and implemented for the ARMv7-A architecture and, currently, even more specifically for Cortex-A8 processors. Despite this specifity, SobTrA provides the basis for an implementation on other ARM cores. For multi-core processors SobTrA could be combined with the generic approach by Yan et al. [19]. The ARMv7-A architecture has some specialties which must

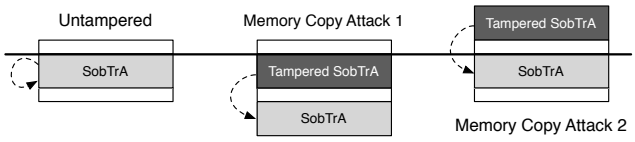

Figure 5: Two forms of memory copy attacks

```
1:  Input:
        y: number of iterations of the function
        challenge initializing C, rand and r13
2:  Output: Checksum C
3:  Variables:
        [start_code,end_code]: checksummed memory area
        daddr: address of current memory access
        rand: current pseudorandom number
        r13: intermediate result from previous iteration
        t: variable to save parallel processing results
        j: index of current checksum part
        l: loop counter
        SR: status (flags) register
        PC: program counter
4:  for l = y to 0 do
5:      // T-Function [10] updates rand
6:      rand ← rand + (rand^2 ∨ 5) mod 2^32
7:      // Update memory address with rand
8:      daddr ← ((C_{j-1} ⊕ rand) ∧ MASK) + start
9:      // Update checksum part with index j
10:     C_j ← C_j + PC
11:     t ← mem[daddr]
12:     C_j ← t ⊕ rotate(C_j)
13:     t ← t + r13
14:     r13 ← r13 + C_j
15:     t ← t ⊕ l + C_{j-1} ⊕ rand + daddr ⊕ C_{j-2}
16:     C_j ← C_j + PC ⊕ l +_c C_{j-1} ⊕ rand +_c daddr ⊕ C_{j-2} +_c t
17:     C_j ← SR ⊕ rotate(C_j)
18:     // Update index j
19:     j ← (j + 1) mod 10
20: end for
```

Figure 6: SobTrA checksum function algorithm

be considered in the checksum function design: First, the Program Counter (PC) is represented as an explicit register while the Status Register (SR) is *not* accessible like a normal register and flags must be set explicitly. Second, memory accesses are always done by dedicated instructions (load/store architecture). Third, many instructions allow the second operand to be rotated/shifted without an additional instruction (and without costs at least on Cortex-A8). Finally, there are two different instruction sets with different instruction encodings: ARM and Thumb–2. The latter mixes 16 and 32 bit wide instructions for higher code density.

It is crucial that the hardware is configured to provide the maximum possible performance for running the checksum function. Every unused optimization could potentially be abused by an attacker to hide latencies introduced by his code. So, the CPU must be configured to run the highest possible clock speed available in software and branch prediction as well as all caches must be activated. Latter requires the Memory Management Unit (MMU) to be *enabled* [3].

Besides these performance initializations, it is also necessary to temporarily disable all *interrupts* to prevent an attacker from gaining control during or shortly after the checksum function. Our implementation includes all these initializations. An attacker cannot change these settings without reducing performance and therefore failing a verification.

5.1 Basic Algorithm

Figure 6 shows pseudocode for the SobTrA checksum function main loop. The loop body can be structured into three parts:

1. Calculation of the next pseudorandom number, i.e., the *pseudorandom update* (Line 6).

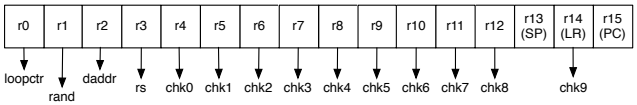

Figure 7: Mapping of ARM registers to their names in the SobTrA checksum function

2. Calculation of the address of the next memory word to be incorporated into the checksum, i.e., the *data address update* (Line 8).

3. Update of the current checksum part (Lines 10 to 17).

Together with a final loop counter decrementation, these three parts are called *basic block* or *block* in the following. The checksum is split into several parts to fill the available registers. Part 2 and 3 of the algorithm are specific to the checksum part updated in the current iteration. The checksum update algorithm part is highly architecture specific and the design ideas will therefore be explained later together with the implementation. Most of the previously introduced checksum function properties, for example the pseudorandom memory traversal and CPU state inputs (PC and SR), are fulfilled obviously. The function is strongly ordered and non-parallelizable by making blocks dependent on previous results, interdependencies between subparts of the loop body and by interleaving add +, add-with-carry $+_c$ and xor ⊕ operations for updating the checksum parts.

5.2 Main Loop Implementation

The checksum function implementation runs in Supervisor Mode. The presented assembler code uses named registers, as specified in Figure 7, to increase readability. There is one dedicated register for each running variable in the algorithm and an additional scratch register named `rs` to hold different values, especially the t variable, during a loop iteration. Ten registers (`chk0` to `chk9`) store the checksum parts resulting in a 320 bit wide checksum. To make most efficient use of available registers, the algorithm loop is *unrolled* in ten checksum part specific blocks, so the index j in the pseudocode has no equivalent in the actual implementation. The SobTrA checksum function implementation must be encoded using the Thumb–2 instruction set because the checksum function leverages the mixed instruction size (16 and 32 bits) to prevent a specific form of memory copy attack (discussed in Section 5.3.2). In Thumb–2 encoding, the registers `r13` (Stack Pointer) and `r15` (PC) are restricted to be usable only in some instructions. Register `r13` can therefore not be used as a checksum register or working register. To fulfill the property of using all registers, `r13` is occupied by saving an intermediate result to be included into the checksum in the next block. So it maps directly to the identically named variable in the algorithm pseudocode. To reduce the possibilities for an attacker to insert instructions without overhead, we did a manual cycle-exact analysis of the checksum function implementation for the Cortex-A8 dual-issue, in-order pipeline using information provided by [2]. Based on the analysis (verified by performance counters) we made sure that the presented implementation makes highly efficient use of the pipeline features, so that a basic block executes in only 31 processor cycles. The branch instruction at the end of the unrolled loop and the self-modification sequence added in Section 5.3.3 account for another 0.1 and 0.42 cycles per block.

```
1 mul rs, rand, rand          @ x^2
2 orr rs, rs, 0x5             @ x^2 ∨ 5
3 add rand, rs, rand          @ (x^2 ∨ 5) + x
```

Figure 8: Pseudorandom generator update sequence

The checksum function is **initialized** with a 320 bit wide challenge filling all checksum registers with a starting value. The seed for the random number generator is derived from the challenge by xoring their parts. Besides that, also `r13` is initialized with some value derived from `rand`.

Like other concepts [16, 11, 15], SobTrA uses a T-Function [10] for the **pseudorandom update** (Figure 6, Line 6), mainly because it can be implemented with only few instructions as shown in Figure 8. Therefore, an optimal implementation can be ensured more easily. The calculation only depends on the previous value of `rand` and uses the scratch register `rs` to hold intermediate results.

The SobTrA **data address update** implementation for the first checksum function block is shown in Figure 9. For optimal performance, only word-aligned addresses are generated. Eight KiB beginning at the `start_code` label are checksummed, which is enough to include all SobTrA parts. The `start_code` label address is generated PC-relative using the `adr` instruction.

Figure 10 shows the SobTrA implementation of the **checksum update sequence**. The code updates the first checksum part register and is therefore specific to the first checksum function block of the unrolled loop. To keep the scratch register `rs`, used for loading the memory value (Line 10), alive and unavailable to an attacker, SobTrA processes it with a sequence of operations parallel to the checksum register and incorporates it into the checksum just before it is needed again to load the Current Program Status Register (CPSR) (the ARM status register). These operations fill the Cortex-A8 dual-issue pipeline optimally and therefore cause nearly no overhead. Some operations are used in their flag-setting variants (mnemonic suffixed with "s") to impede the forgery of the CPSR. This is particularly important for the addition immediately before the `mrs` instruction, which reads the CPSR, since it prevents reordering of this instruction to an upper position. The `mrs` instruction is quite expensive, costing about eight CPU execution cycles on the Cortex-A8 [2]. Nevertheless, it is not possible to forge the CPSR in the same or less time since an attacker would have to forge the dynamically set flags. To keep register `r13` unavailable to an attacker, SobTrA uses two instructions (Lines 12 and 13) which incorporate an intermediate result from the previous block into the checksum. To reduce the overhead, to maintain the order of the `add`/`eor` sequence and to harden against certain memory copy attacks (discussed later), this is done via an indirection over the scratch register (i.e., variable t) which is included into the checksum near the end of the sequence (Line 25). Further details of the implementation are discussed in the next section with regard to the specific attacks they prevent.

```
4 eors daddr, chk9, rand @ derive random value
5 ldr  rs, =0x7ff<<2     @ load mask in scratch register
6 and  daddr, daddr, rs  @ mask random to get an offset
7 adr  rs, start_code    @ gen. start address PC-relative
8 add  daddr, daddr, rs  @ add offset to start address
```

Figure 9: SobTrA data address update implementation for the first block

```
9  add  chk0, chk0, pc          @ C_0 ← C_0 + PC
10 ldr  rs, [daddr]             @ t ← mem[daddr]
11 eor  chk0, rs, chk0, ror 1   @ C_0 ← t ⊕ rotate(C_0)
12 add  rs, rs, r13             @ t ← t + r13
13 add  r13, chk0               @ r13 ← r13 + C_0
14 add  chk0, chk0, pc          @ C_0 ← C_0 + PC
15 eors rs, rs, loopctr         @ t ← t ⊕ l
16 eors chk0, chk0, loopctr     @ C_0 ← C_0 ⊕ l
17 add  rs, rs, chk9            @ t ← t + C_9
18 adcs chk0, chk0, chk9        @ C_0 ← C_0 +_c C_9
19 eors rs, rs, rand            @ t ← t ⊕ rand
20 eors chk0, chk0, rand        @ C_0 ← C_0 ⊕ rand
21 add  rs, rs, daddr           @ t ← t + daddr
22 adcs chk0, chk0, daddr       @ C_0 ← C_0 +_c daddr
23 eors rs, rs, chk8            @ t ← t ⊕ C_8
24 eors chk0, chk0, chk8        @ C_0 ← C_0 ⊕ C_8
25 adcs chk0, chk0, rs          @ C_0 ← C_0 +_c t
26 mrs  rs, cpsr                @ load the SR
27 eor  chk0, rs, chk0, ror 1   @ C_0 ← SR ⊕ rotate(C_0)
28 sub  loopctr, loopctr, 1     @ decrement l
```

Figure 10: SobTrA checksum update sequence (first block)

5.3 Attack Prevention

In the following, attacks on the checksum function are discussed and how SobTrA prevents them. With one block taking 31.52 cycles to execute, an overhead of only one CPU cycle per block ($^{32.52}/_{31.52} - 1 = 3.17\%$) is enough to be easily measurable (see Section 7). Therefore, it is sufficient to show that an attack generates an overhead of at least one cycle to be prevented successfully.

5.3.1 Substitution Attacks

A successful substitution attack replaces parts of the checksummed memory region without changing the resulting checksum or causing an execution time overhead. The most basic forms of substitution attacks are prevented by the pseudorandom memory traversal of the algorithm, which forces an attacker to insert, at least, a basic conditional (`if`-like) statement into the checksum loop to check if a location previously modified by him is read and to hide the modification accordingly. This easily causes an overhead of at least one cycle. Like Seshadri et al. [14] we use the *Coupon Collector's Problem* to calculate the number of iterations needed to include every memory word at least once with high probability. For 8 KiB (equaling 2048 32-bit words) this results in $2048 \ln 2048 + 2048 \approx 17663$ iterations.

Castelluccia et al. introduced an attack [4] which exploits the fact that the Most Significant Bit (MSB) is treated the exact same way by an addition as by a bitwise exclusive-or. Therefore, a sequence of alternating additions and exclusive-ors, like it is the basis for updating the checksum, yields the same result if the MSB is inverted in an even number of operands. In order to prevent such attacks, SobTrA implements, like other concepts [12], flag-setting add-with-carry (`adcs`) operations instead of simple additions.

Unfortunately, Thumb–2 does not allow the PC as operand to any add-with-carry operation. This results in an attack where the MSBs of the PC in the two PC incorporating instructions, in the following called *PC inclusions* (Figure 10, Lines 9 and 14), cancel each other out. So a SobTrA execution from a location with inverted MSBs would result in the same checksum. SobTrA prevents such an attack by rotating the checksum between both PC inclusions (Line 11). Since the rotation is piggybacked by the `eor` instruction, it does not cost additional cycles on the Cortex-A8 processor.

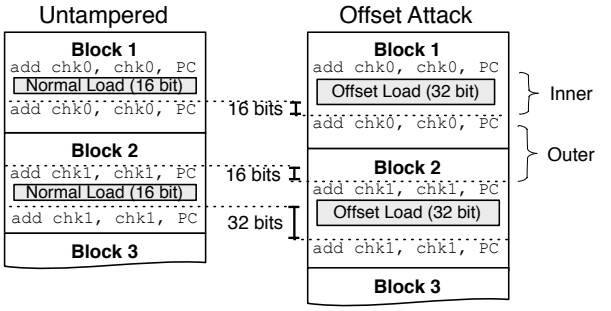

Figure 11: PC corruption for the offset attack on the checksum function

Furthermore, SobTrA prevents MSB attacks spanning over multiple iterations of the unrolled loop with a rotation at the end of every block (Line 27) and the address update being dependent on previous block's checksum register.

5.3.2 Memory Copy Attacks Type 1

In memory copy attacks of this type, the attacker runs his malicious checksum function in the originally intended place, but computes the checksum over an untampered image stored somewhere else in memory (see Figure 5 in the middle). Since the ARM architecture allows a load instruction to have a fixed offset without additional costs, such an attack could be realized quite easily by replacing the load instruction with an offset variant. For a successful *offset attack*, the untampered image has to be in range of the load offset. This range is different depending on the used instruction encoding. To prevent the offset attack, the SobTrA checksum function leverages the fact that there are different size encodings of the load (`ldr`) instruction in the Thumb–2 instruction set. There is only one 16-bit encoding (T1), which allows loads with the `daddr` base register. For this encoding, the range of offsets is 0-124 in multiples of four. This is *not* sufficient for a successful attack since the unrolled checksum function loop itself is about 1 KiB in size and can therefore not be passed. If the attacker uses a 32-bit instruction encoding which allows offsets to be big enough, he implicitly changes the position of the following PC inclusions and therefore corrupts the checksum. The deviation of the PC inclusions from their original position cumulates for the following blocks in the unrolled loop. Figure 11 illustrates the situation for the attacker. To repair the corrupted PC inclusions, the attacker may either forge the PCs, which in fact leads to a type 2 memory copy attack, or try to balance the bigger offset load instruction by encoding other instructions shorter. To prevent such attacks, we examined the SobTrA code to use the smallest possible encoding (e.g. use `eors` instead of `eor`). Especially in the *inner* region (see Figure 11) this is important since there only 16 bits must be saved for a successful attack. Saving 16 bits in the *outer* region is *not* enough as the distance between the two PC inclusions must be correct as well. Many instructions can only be encoded in their short form (16 bits) if the operands are from the lower half of the register bank (registers 0-7) so SobTrA maps the frequently used variables to lower register.

5.3.3 Memory Copy Attacks Type 2

In this type of memory copy attacks, the untampered SobTrA image is kept in its originally intended position and the attacker checksums it from another position. There are three spots in the checksum function which have to be considered for such an attack because of their PC usage:

- The base address is generated PC-relative (Figure 9, Line 7) and the `adr` instruction must therefore be in range of the untampered code start.
- The **two** PC inclusions (Figure 10, Lines 9 and 14) must be forged. For the first forgery, the scratch register `rs` is available, for the second it is *not*.

The attacker is free in terms of code size and encoding because he forges the PC inclusions anyway. He is even free to use the ARM instruction set instead of Thumb–2 because the `mrs` instruction does *not* read the actual values of the processor state bits, encoding if the processor runs in ARM or Thumb mode [3]. We identified two different type 2 memory copy attacks to be discussed in the following.

Basic PC Forgery. An attacker might try to replace the PC register operand with immediates, i.e., static values to be encoded directly into the instruction. Because of the restrictions on immediates in the ARM architecture, such an attack would be only possible for unrealistic simple addresses. As the place for the checksum function is specified by the verifier, this does not pose a problem.

An attacker might try to forge the PC values with basic arithmetic operations on his own PC. This is only possible if the attacker's code is in range of the untampered code. The first PC inclusion can then be forged using the free scratch register `rs`. The second PC inclusion *cannot* be forged like this because there is no spare register available. So, forging the second PC value forces the attacker to spill a register, i.e., store it to memory and restore it afterwards, easily causing an overhead of at least one cycle. Besides the additional store and load instruction, spilling a register in the most cases incurs overhead for an extra address generation.

Shifting Attack. This attack is based on the idea to forge the original PC only by shifting the actual attacker's PC, exploiting free shifts on Cortex-A8 CPUs. The malicious checksum function is positioned in a way so that its base address equals the base address of the untampered checksum function logically left-shifted by one. Thumb–2 does not allow shifts/rotates on the PC, but the attacker is free to use ARM encoding which allows shifting and adding of the PC in the same instruction. The attacker has to align its code (e.g., by inserting NOPs) so that its shifting PC inclusions are in positions where they yield the correct PC just by right-shifting.

The SobTrA checksum function prevents such an attack by running from addresses which cannot be generated by a simple logical or arithmetic right-shift (possibly configuring the MMU accordingly). The key characteristic of such an address is that its two most significant bits are "10". Furthermore, the attack is prevented because of the PC-relative base address generation in the data address update sequence, requiring the attacker to either do a shift to get in range and subtract an offset (always two instructions), or to do a memory load, both causing at least a one cycle overhead.

5.3.4 TLB Desynchronization Attack

Wurster et al. in 2005 introduced [18] a technique which exploits Harvard-style Translation Lookaside Buffers (TLBs)

to attack self-checksumming code. The basic idea of the attack is to desynchronize the TLBs in a way that the same Virtual Address (VA) is mapped to a malicious copy of the checksum function in the Instruction Translation Lookaside Buffer (ITLB) and to an untampered image in the Data Translation Lookaside Buffer (DTLB). Then, all instructions are fetched from the malicious checksum function in physical memory, but the load instructions, used to checksum the code, access the untampered copy in physical memory, resulting in a quasi hardware-accelerated memory copy attack. We successfully implemented the attack on ARM Cortex-A8 in a less generic but much easier way than described by Wurster et al. leveraging the explicit DTLB invalidation operations available on ARM.

Giffin et al. [6] in 2005 proposed a technique using self-modifying code to strengthen self-checksumming code against TLB desynchronization attacks. Self-modifying code accesses its code via the DTLB, and therefore corrupts the checksum in case of a TLB desynchronization. Like some related work concepts [11, 13] for x86 SobTrA uses self-modifying code to prevent such attacks but employs a different approach since, in contrast to x86, the ARM architecture requires explicit and expensive cache maintenance operations when implementing self-modifying code, mainly to propagate the changes from the data cache to the instruction cache. SobTrA optimizes the self-modification sequence by configuring the level 1 cache to be write-through and the level 2 cache to be write-back and by removing unnecessary barriers. Still, The necessary instruction cache invalidation costs about 60 cycles, which is too long in comparison to a block's execution time of 31 cycles to do a self-modification in every block. Therefore, SobTrA reduces the per-block overhead by doing a self-modification only every n-th block. The self-modification code is executed every $^{n}/_{10}$-th unrolled loop iteration. It randomly picks one block and changes the rotation immediate of a specific instruction (Figure 10, Line 11) in the block to a random value. An attacker who simulates the self-modification functionality in a static way, has to do the same computations every n-th block to update his untampered image with the modified instructions. He saves 60 cycles ($^{60}/_{n}$ per block) from not having to issue the cache line flush, but for the simulation additional instructions are required in *every* attacker's block (e.g., load number of rotations and rotate by register). This simulation causes at least one cycle overhead per block since there is no spare register available and additional instructions corrupt the PC inclusions, especially in the inner region where the simulation has to be done (see Section 5.3.2).

6. SECURE EXECUTION ENVIRONMENT

SobTrA must ensure that an attacker *cannot* gain control after the successful computation of the checksum. Otherwise, such an attacker could use the valid checksum to obtain the trust of the verifier and run arbitrary code afterwards.

Assuming that the checksum function has finished successfully, it is ensured that only checksummed and untampered code is executed afterwards. Therefore, the only possible way for an attacker to gain control of the execution is to trigger an exception. The ARM literature uses the term *exception* to denote all kinds of interrupts or traps and the term *interrupt* to denote only asynchronous interrupts. SobTrA establishes a secure execution environment by replacing the exception vector table with an own table (containing dead

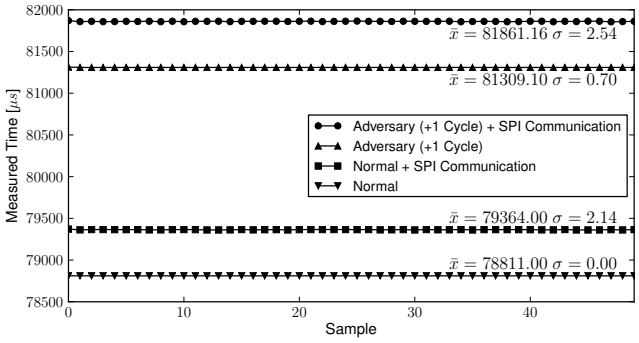

Figure 12: Consecutive measurements of execution time for 1.5 million iterations

loops) stored *inside* the checksummed region immediately after the checksum calculation and *before* sending the result. Code inside the checksummed-region may *not* cause any exceptions or call unmeasured and unchecksummed code.

It must also be ensured that an attacker cannot gain control *before* the secure execution environment is established, i.e., during and shortly after the checksum function execution. For that, we analyzed the different exception types regarding their attack potential. As shown before, it can be assumed that there are no successful attacks on the checksum function itself, i.e., it is guaranteed that the SobTrA code runs untampered:

Undefined Instruction and Supervisor Call. Synchronous exceptions are caused explicitly by instructions. Since the SobTrA code runs untampered and does not trigger any of these, they do not pose a threat.

IRQ, FIQ and external Abort. External interrupts are disabled by SobTrA and their status is incorporated into the checksum via the CPSR.

Prefetch Abort. These are triggered by the MMU when fetching instructions from invalid memory addresses. The checksum function and secure execution environment establishment code reside on the same minimal page, which prevents Prefetch Aborts.

Data Abort. These are triggered by the MMU when accessing invalid data addresses. Since the checksummed region is about 8 KiB in size, a Data Abort might be caused by an attacker *during* the checksum calculation. As the calculation is not yet finished when such an exception might occur and every execution of the exception takes long in relation to one checksum function block, it cannot be abused by an attacker.

7. EXPERIMENTAL RESULTS

We implemented a prototype, which is able to run a complete software-based secure boot using the SobTrA primitive. The prototype hardware consists of a Beagleboard (Rev. C3, Cortex-A8, 600 MHz) as the untrusted platform and a mbed LPC1768 microcontroller as verifier, interconnected via SPI. SobTrA on the Beagleboard is implemented as bare-metal, standalone binary image containing the checksum function, an SPI driver for the communication protocol and a minimal ARM Linux kernel bootloader as well as a

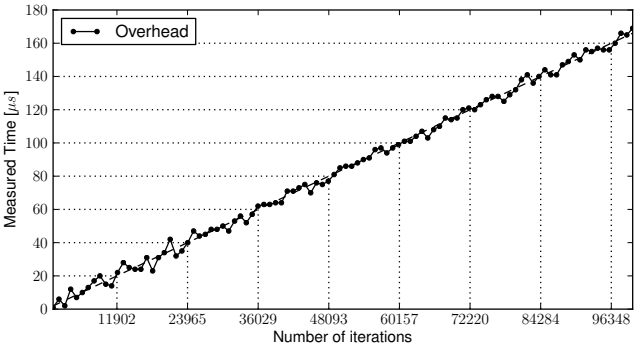

Figure 13: Measured attacker overhead for 1000 to 100000 iterations

minimal SHA-1 implementation to hash and start a trusted kernel. To support arbitrary hardware as verifier, we implemented a SobTrA simulator in C which is used to compute challenge-response pairs on the mbed LPC1768.

First, we measured the SobTrA execution time several times with the same number of iterations in the untampered case and in case of a minimal attack causing a *one* cycle overhead per checksum function block. For each case, we differentiated pure execution time and time measured by the verifier via SPI at 3 MHz. The resulting plot is depicted in Figure 12 with mean value ($\bar{x}$) and standard deviation (σ). With an overhead of $^{81861}/_{79364} - 1 \approx 3.15\%$ for the communication including case and the low latency of the SPI bus, even attacker overheads smaller than one cycle are measurable quite well.

In a second step, we measured the execution time, including communication overheads, for an increasing number of iterations. Figure 13 shows the overhead generated by the attacker (one cycle overhead), i.e., the difference between normal and tampered execution time. On the average, about 24000 iterations are needed to generate 40 µs overhead for the attacking case, sufficient to be reliably measured. The 24000 iterations are also sufficient to cover SobTrA's 8 KiB in the random memory traversal (see Section 5.3.1). The SobTrA execution time in this case is about 1.8 ms.

8. CONCLUSION

In this paper, we introduced SobTrA, a software-based trust anchor for ARM Cortex-A processors. As our main application scenario, the concept of a software-based secure boot for embedded devices, e.g., smartphones without usable hardware-based secure boot, was presented. In this concept, a verifier device attached to an untrusted platform obtains the guarantee that the boot process on the untrusted device is running untampered. The checksum function provides the guarantee to the verifier that SobTrA runs untampered by performing an optimized, self-checksumming calculation in a certain, upper-bounded timeframe.

We implemented a Cortex-A8 specific checksum function and showed that it is resistant against known attacks. Furthermore, we implemented a secure execution environment, preventing attacker interception by making sure that no exceptions can be triggered during the checksum calculation and replacing the exception vector table afterwards. Our prototype is able to run a software-based secure boot up to a Linux kernel. Experiments revealed that the approach is very robust in terms of timing, making it possible to clearly distinguish untampered and tampered execution even for very small overheads introduced by an attacker.

9. ACKNOWLEDGMENTS

Parts of this work were supported by the German Federal Ministry of Education and Research (BMBF) under grant 01BY1200A within the project *HIVE*.

10. REFERENCES

[1] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997.

[2] ARM Limited. *ARM Cortex-A8 Technical Reference Manual*, May 2010.

[3] ARM Limited. *ARM Architecture Reference Manual – ARMv7-A and ARMv7-R edition*, July 2011.

[4] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, 2009.

[5] R. W. Gardner, S. Garera, and A. D. Rubin. Detecting code alteration by creating a temporary memory bottleneck. *Trans. Info. For. Sec.*, 4:638–650, December 2009.

[6] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 23–32, 2005.

[7] M. Jakobsson and K.-A. Johansson. Practical and secure software-based attestation. In *Workshop on Lightweight Security Privacy: Devices, Protocols and Applications (LightSec)*, pages 1–9, March 2011.

[8] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 304–314, 2002.

[9] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[10] A. Klimov and A. Shamir. New cryptographic primitives based on multiword T-Functions. In B. Roy and W. Meier, editors, *Fast Software Encryption*, volume 3017 of *Lecture Notes in Computer Science*, pages 1–15. 2004.

[11] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: Tamper-proof code execution on legacy systems. In *Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment*, DIMVA'10, 2010.

[12] A. Perrig and L. V. Doorn. Refutation of "On the difficulty of software-based attestation of embedded devices".

[13] A. Seshadri. *A software primitive for externally-verifiable untampered execution and its applications to securing computing systems*. PhD thesis, Carnegie Mellon University, 2009.

[14] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security*, WiSe '06, pages 85–94, 2006.

[15] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 1–16, 2005.

[16] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 272–282, 2004.

[17] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th conference on USENIX Security Symposium*, SSYM'04, 2004.

[18] G. Wurster, P. C. v. Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 127–138, 2005.

[19] Q. Yan, J. Han, Y. Li, R. H. Deng, and T. Li. A software-based root-of-trust primitive on multicore platforms. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, 2011.