

# Segregating Keys from *noncense*: Timely Exfil of Ephemeral Keys from Embedded Systems

Heini Bergsson Debes\*, Thanassis Giannetsos†

Technical University of Denmark (DTU), Cyber Security Section, Denmark

‡Ubitech Ltd., Digital Security & Trusted Computing Group, Greece

Email: heib@dtu.dk, agiannetsos@ubitech.eu

**Abstract**—As lightweight embedded devices become increasingly ubiquitous and connected, they present a disturbing target for adversaries circumventing the gates of cryptography. We consider the challenge of exfiltrating and locating cryptographic keys from the run-time environment of software-based services when their software layout and data structures in memory are unknown. We detail an attack that can, without affecting the system’s operation, exfiltrate keys *in use* promptly by leveraging the strong causality between transceivers and keyed cryptosystems (authentication, authorization, and encryption). We then propose how to effectively and efficiently reduce the key material’s search space from a batch of stackshots (stack extractions) by leveraging the stack’s innate composition, which, to the best of our knowledge, is the first method to systematically infer and reduce the search space of semi-arbitrary keys. We instantiate and evaluate our attack against MSP430 micro-controllers.

**Index Terms**—Key-Exposure Problem, Runtime Key Disclosure

## I. INTRODUCTION

With the advent of the Internet of Things (IoT), deployment of embedded systems has accelerated, and systems have become unprecedentedly network-connected, autonomous, and collaborative (e.g., smart homes, automotive, healthcare, agriculture, industrial). However, in current security, privacy, and safety-critical IoT application domains, a significant portion of *resource-constrained* microcontroller-based embedded systems (MCUS) have the inherent lack, due to cost, of tamper-proof hardware or essential protection mechanisms found in their desktop counterparts, e.g., Cryptographic Modules (CM), Data Execution Prevention (DEP), stack canaries, and Address Space Layout Randomization (ASLR) [1], [2], leaving MCUS susceptible to an increasing number of remote attacks [3].

Coupled with resource-constrained Operating Systems (OS, e.g., Contiki, FreeRTOS, or TinyOS [4]) or bare-metal (without any OS), MCUS run single binary images where the application orchestrates all system resources and, due to the lack of onboard protection mechanisms, determines the system’s security posture, which, generally comprises keyed authentication, authorization, or encryption cryptosystems. However, such cryptosystems’ security lies solely in the secrecy of the *secret* key (Kerckhoffs’s principle); so does that of MCUS.

To retain key secrecy in *all* stages (creation, dissemination, storage, and *usage*) is non-trivial. Computing systems inherently require that any program, including its data and instructions, be loaded into the main memory before being

run by the processor. Thus, while keys can be protected while stored [5], any conventional program that performs keyed cryptographic operations must, at some point, have the key material exposed [6] (known as the *Key-Exposure Problem*, KEP). The KEP might not be a *problem* if we assume that key management is only performed behind closed curtains. However, since adversaries *are* able to infiltrate MCUS [3], we must opt to consider adversaries peeking behind said curtain.

While there exist ways to get a peek [7]–[9], the inherent deficiency lies in assuming that keys are visible in memory at acquisition time. Until now, attempts to narrow on the timeliness [10], [11] or locating (or reduce the search space of) key material once the memory is obtained [7], [8], [12]–[14] have been inherently cryptosystem- or software-dependent.

**Contributions:** We demonstrate how adversaries can, *without knowing the software layout or memory data structures of running services*, exploit the KEP in *network-connected* MCUS to exfiltrate cryptographic keys, during system operation systematically, *without* affecting system usability. We present *generic* methods for overcoming two significant challenges revolving around *successful* key exfiltration: (i) how to acquire the memory contents systematically *while* the key is exposed (exfiltration phase) and (ii) how to efficiently reduce the search space of arbitrary key material (localization phase). Specifically, targeting the nature of MCUS, we demonstrate how to exploit the causality between transceiver invocation and utilization of keyed cryptosystems to acquire timely memory extractions. Concretely, since keyed cryptosystems inherently run post-reception (e.g., to verify or decrypt an incoming payload), we can, by periodically exfiltrating conventionally used memory regions for storing run-time data (e.g., the memory *stack*), capture data belonging to the keyed cryptographic function during the inevitable Key Exposure Window (KEW) caused by the KEP. Further, as a software- and cryptosystem-agnostic method of locating key material, we propose to apply specific data mining techniques (Section V-C) that leverage the stack’s innate composition. The intuition behind the presented work is to showcase how vulnerable the existing commodity MCUS are against sophisticated attacks and emphasize the need for appropriate prevention strategies (Section VIII).

## II. RELATED WORK: TOWARD KEY IDENTIFICATION

In 1998, after observing the inherent randomness in cryptographic keys, Shamir et al. [7] postulated that memory regions

with unusually high entropy might infer a key's presence. However, their conjecture that keys have higher entropy than other data is not always valid [8]. The approach becomes even less attractive when considering symmetric keys, which are conventionally much smaller than asymmetric keys. For symmetric keys, Halderman et al. [8] proposed searching for mathematical properties of the AES key schedule and further conceptualized a semi-unified search for different types of keys (symmetric and asymmetric) by incorporating heuristics about the cryptographic algorithms, e.g., well-known RSA encodings wherein the key is encapsulated within fixed structures [12] or the memory reflections of code structures containing key material in standardized implementations [13]. The inherent deficiency of each approach is that it is viable only for a target algorithm or is applicable *only when implementations yield the presumed structural representation in memory*. For example, although the AES key schedule has distinctive characteristics, it cannot be said about symmetric keys in general as randomness is their only definite identifiable characteristic. Further, the schedule's structural reflection in memory is also *not* fixed *nor* certain. Such issues make any attempt toward unified key localization from memory contents *extremely difficult*.

To narrow the search space on which key identification is conducted, the authors of [14] propose reconstructing call stacks of functions that invoke security-sensitive Windows APIs, known to accept keys as arguments. The premise (which is also used in this paper) is that program functions generally use the memory stack to hold variables during execution, and thus the stack of keyed functions will inevitably contain key materials. However, besides being highly application dependent, in practice, compiler optimizations will severely complicate call stack reconstruction (e.g., its memory reflection and which elements occur) [15], [16]. Further, as they note, there is no guarantee that a key is in memory at the acquisition time.

Considering *timeliness*, authors of [11] target Ransomware keys by monitoring invocations of specific cryptographic APIs to *trigger* memory extraction and authors of [10] monitor the control-flow of network-related functions in Android applications to *trigger* acquisition of TLS key materials. However, whereas they require direct control-flow monitoring capabilities, we propose a more lightweight and application-agnostic trigger heuristic. Further, where they employ algorithm- and implementation-aware key identification techniques, we propose novel exploitation of the program stack's nature to isolate areas likely to contain *whichever* key that might be present.

### III. PROBLEM STATEMENT

The strength of a cryptosystem, with key-length  $kl$ , is quantified by its ability to resist brute-force attacks on the entire key-space  $2^{kl}$ , which *should* be computationally infeasible for large  $kl$ 's. However, applying an exhaustive search on a considerably *reduced* search space of memory is appealing and, in theory, more efficient, as the search space reflects all possible key-sized blocks of contiguous bytes. Let  $D$  denote a device with volatile memory  $VM$ . The shared memory space is defined as  $SM = \{s, r, h\} \in VM$ , comprising the stack  $s$ ,

registers  $r$ , and the heap  $h$ , respectively. Let  $P$  be a program running on  $D$  which uses  $SM$  as its execution environment. For simplicity, assume that  $SM_t = \langle s_t || r_t || h_t \rangle$  is the finite sequence of bytes stored in  $SM$  at time  $t$  and  $|SM_t|$  denotes the cardinality. Let  $SS = \{SM_t : t \in T \subseteq \mathbb{N}^*\}$  denote the search space comprising all instances of  $SM$  in  $D$ 's universal time space  $\mathcal{T}$ . Further, let  $\mathcal{K}$  be a key used by  $P$  such that  $\mathcal{K}$  is stored in its *entirety somewhere* in  $SM$  at times  $T^{\mathcal{K}} \subseteq \mathcal{T}$ .

Let  $\text{Ext} : T \times \mathbb{N}^* \times \mathbb{N}^* \rightarrow SS$  be a three-input extractor function accepting a time  $t$ , a start index  $i$ , and a range  $rg$  to return a subsequence from  $SM$  at time  $t$ . It is defined by  $\text{Ext}(t, i, rg) = \langle a_i, a_{i+1}, \dots, a_{i+rg} \rangle \sqsubseteq SM_t$ , where  $1 \leq i < i + rg \leq |SM_t|$  and  $\sqsubseteq$  denotes *subsequence of*. Hence,  $\forall t \in \mathcal{T}$  we define a map instance  $\text{Ext}_t : \mathbb{N}^* \times \mathbb{N}^* \rightarrow SS$  by  $\text{Ext}_t(i, rg) = \text{Ext}(t, \langle a_i, a_{i+1}, \dots, a_{i+rg} \rangle)$ .

We formalize the search space reduction ( $SSR$ ), where we consider an adversary ( $\mathcal{A}$ ) that, based on select subsequences from instances of  $\text{Ext}$ , tries to find the fewest possible candidates for  $\mathcal{K}$ . The  $SSR$ -complexity is defined as the number of candidates and is over the choices of  $t$ ,  $rg$ , and any choice of  $\mathcal{A}$  herself. Accordingly,  $\mathcal{A}$  is given oracle access to  $\text{Ext}$  so she can obtain subsequences of her choice and is not constrained concerning the method she uses, leading to Definition 3.1.

**Definition 3.1:** Let  $B$  be a  $SSR$  algorithm that takes the function map  $\text{Ext}$  and yields a set of subsequences (*candidate keys*). Considering the experiment in Algorithm 1, the  $SSR$ -complexity of  $B$  is defined as:  $\text{CXTY}_{\text{Ext}}^{\text{SSR}}(B) = \text{Exp}_{\text{Ext}}^{\text{SSR}}(B)$ .

---

#### Algorithm 1: EXPERIMENT $\text{Exp}_{\text{Ext}}^{\text{SSR}}(B)$

---

**Input :**  $SSR$  algorithm,  $B$   
**Output:** Cardinality of the set of candidate keys  
1  $\sigma \leftarrow B^{\text{Ext}}$   
2 **if**  $\mathcal{K} \in \sigma$  **then return**  $|\sigma|$  **else return**  $\perp$

---

The definition is made general enough to capture all types of key-localization attacks. For example, performing a Variable Sliding-Window (VSW, i.e., *linear scan* [17]) attack over the entire contents in  $SS$ , using window sizes  $n = 1, \dots, N \in \mathbb{N}^*$ , yields an upper bound of:  $\text{CXTY}_{\text{Ext}}^{\text{SSR}}(\text{VSW}) = \sum_{n=1}^N |SM| - n + 1, \forall SM \in SS$ . This motivates our question: can  $\mathcal{A}$  reduce the search space more efficiently (i.e., reduce the number of candidate keys) by applying heuristics such as data mining, logical reasoning, and inference? However, note that because the heap is rarely used in resource-constrained MCUS [18], it is not considered. Also, although registers generally contain valuable information, we argue that they are less likely to contain keys in resource-constrained MCUS (Section V-B). Therefore, for this paper, the stack is of *exclusive* interest.

### IV. SYSTEM AND ADVERSARIAL MODEL

We consider how an *software-oblivious*  $\mathcal{A}$  can methodically acquire ephemeral cryptographic keys (i.e., keys that are non-existent before use and securely sanitized immediately afterward) used for authentication, authorization, and encryption in remote resource-constrained MCUS, following the four

attack phases depicted in Fig. 1. We assume that  $\mathcal{A}$  can: (i) access the shared memory, (ii) periodically transmit select shared memory back to herself, and (iii) validate guesses for arbitrary  $\mathcal{K}$ . Note that while acquiring remote code execution is complementary to our work, we stress that  $\mathcal{A}$  can achieve this advantage through a wide set of attack vectors, e.g., through code injection [18] or Return-Oriented Programming (ROP) [19] by exploiting software vulnerabilities [3]. As a running example, we consider the vulnerable reception handler in Fig. 2, where the infamous C *strcpy* function is exploited to unboundedly overwrite a *stack resident* buffer (the variable *c*), causing the stack frame’s return address to point to malicious code (malcode) controlled by  $\mathcal{A}$  as demonstrated in [18]. Finally, although the software running on the device is a black box to  $\mathcal{A}$  (compelling  $\mathcal{A}$  to resort to *software-oblivious-centric* approaches that do not necessitate the source code), she is given knowledge about the target system’s specifications, enabling  $\mathcal{A}$  to leverage publicly accessible documentation. As a case study, the prominent MoteIV Tmote Sky module [20], which has an MSP430-F1611 MCU [21], serves as our target.

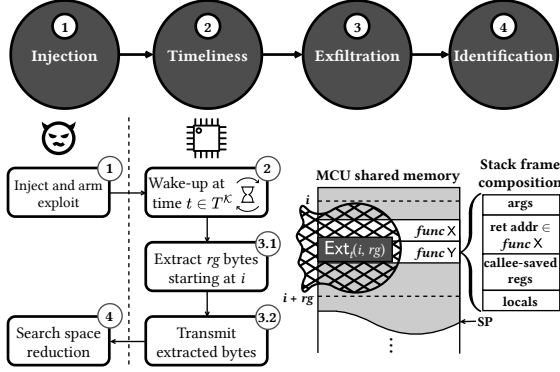


Fig. 1: The four fundamental phases of key acquisition.

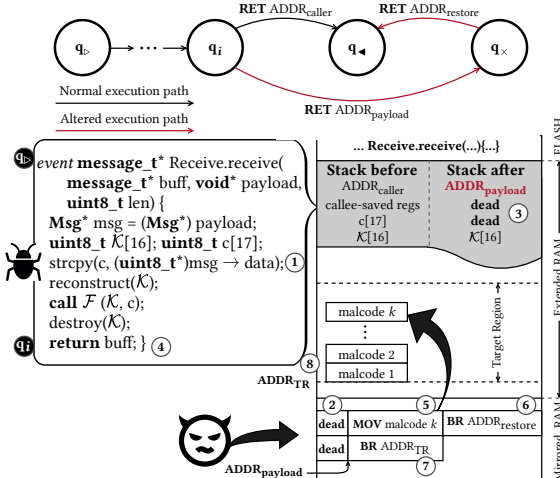


Fig. 2: Multistage code injection through buffer-overflow.

## V. PRELIMINARIES AND METHODOLOGY

### A. Inferring the Imminence of Key Exposure

Oblivious of when keys are used,  $\mathcal{A}$  would have to consider all times as equally likely. However, since the predominant

function of *network-connected* MCUS is data *transmission* and *reception*, some software function will inevitably be invoked to process incoming transmissions (e.g., to decrypt and verify). Thus, if  $\mathcal{A}$  exploits the reception event to trigger  $\mathcal{A}$ -controlled code,  $\mathcal{A}$  can effectively align stackshots to the inevitable KEW.

Since reception involves a transceiver peripheral,  $\mathcal{A}$  can, in a *semi-software-oblivious* manner, exploit it. Specifically, transceivers (as other peripherals) will send signals on designated MCU pins to interrupt the application Central Processing Unit (CPU) about events, where, according to an Interrupt Vector Table (IVTBL), an Interrupt Service Routine (ISR) will run (e.g., to maneuver incoming data into application memory). Thus, given the MCU specifications,  $\mathcal{A}$  can identify which IVTBL entry holds the reception ISR’s memory address.

Note that despite the ability to locate an ISR’s starting point, it is improbable to locate and traverse call-chains leading up to OS functions in a *software-oblivious* manner. Specifically, while  $\mathcal{A}$  can locate the reception ISR’s starting address which causes invocation of the *receive* function in Fig. 2, the concrete ISR offset pointing to the call-chain entry remains unknown, prohibiting  $\mathcal{A}$  from systematically attacking the function. Note, however, if  $\mathcal{A}$  was unbounded, she could flush out the entire memory, attempt to deobfuscate the executable code, and then devise a software-dependent attack to acquire the key material post-reconstruction. However, since we aim for a stealthy, generic, and systematic method, such an attack is not in  $\mathcal{A}$ ’s favor. Nevertheless, if  $\mathcal{A}$  inserts a callback to commence periodic stackshots into the deterministically located reception ISR’s prologue, stackshots will inevitably occur close to the KEW, assuming the causality between reception and keyed cryptosystems holds, *regardless* of the software underpinnings. Additionally, since most MCUS use the Memory-Mapped I/O (MMIO) paradigm for bilateral peripheral communication, another approach to align stackshots with the KEW is to identify and poll memory locations reflecting peripheral states. We consider both approaches to achieve timeliness in Section VI.

### B. Inferring the Key’s Presence

We proceed to infer the whereabouts of cryptographic keys in MCUS. Although data can theoretically reside anywhere (i.e., CPU registers, stack, heap, or other addressable memory regions), in practice, its placement is constrained by several factors, e.g., system resources, compiler, and adherence to the system’s Application Binary Interface (ABI). As a running example, consider a reception handler that accepts and applies some arbitrary cryptographic function  $\mathcal{F}$  on a packet’s contents *c* using the key  $\mathcal{K}$  (materialized using *reconstruct*). The general uses of  $\mathcal{K}$ , which affect its memory placement, are illustrated in Fig. 3 and include:  $\mathcal{K}$  is declared and initialized as a function variable and passed by reference to  $\mathcal{F}$  (use case #1);  $\mathcal{K}$  is passed by value to  $\mathcal{F}$  (use case #2);  $\mathcal{K}$  is declared *above* function level and passed by reference to  $\mathcal{F}$  (use case #3).

Since general-purpose CPU registers are inherently scarce in MCUS, keys are rarely eligible to be confined into registers. For example, the MSP430-F1611 has only 12 16-bit registers and prohibits objects exceeding 64 bits from occupying any

USE CASE #1	USE CASE #2	USE CASE #3
<code>void recv(uint8_t*c){   uint8_t K[16];   reconstruct(K);   F(K, c); }</code>	<code>struct S{   uint8_t K[16];   void recv(uint8_t*c){     F(reconstruct(), c); }</code>	<code>void recv(uint8_t*c){   reconstruct(K);   F(K, c); }</code>

Fig. 3: The key ( $K$ ) is generally stored on the stack when it is a local variable or passed by value (#1 and #2) but not when it is a global variable and passed by reference (#3).

registers [21]. Thus, considering the relatively small 16-byte  $K$  (Fig. 3), it inevitably resides on the stack in use cases #1 and #2. However, because  $K$  is declared *above the function level* in use case #3,  $K$  does *not* occur explicitly on the stack. Instead,  $K$  is put together with similarly scoped variables in a separate RAM region while its *reference* is passed to  $F$ , either in registers or on the stack. Note, however, that although we target the stack (Section III),  $\mathcal{A}$  could mitigate the uncertainty revolving around explicit and referenced keys by regarding each word on the stack as a potential reference and substitute it with a portion from the referenced memory if it references addressable memory. However, as words of a key might also resolve to addressable memory, additional care must be done. Nonetheless, for brevity, we assume that keys occur explicitly.

a) *The stack*: is an aggregate of several consecutively allocated stack *frames* (see Fig. 1). Each program function is appointed one frame, which serves as its scratch space during execution, and contains arguments, CPU state (registers), local variables and temporaries, and a return address, which for callees, points to an instruction that resumes the caller. As functions push and pop elements, the Stack Pointer (SP) continuously points to the stack’s top (most recent element).

Despite  $\mathcal{A}$ ’s inability to know  $K$ ’s definite stack placement, she can approximate it. Since cryptographic functions are inherently designed as leaf functions to mitigate key propagation issues [16],  $K$  likely remains within proximity of the SP during  $F$ ’s execution. For example, in use cases #1 and #2,  $K$  occurs in either the current ( $F$ ) or preceding (*recv*) stack frame. Thus, regarding effective use of Ext (Section III),  $\mathcal{A}$  has, besides *timeliness* (Section V-A), a start index (SP), and justification that a short *range* (*rg*) can suffice for small cryptographic keys.

### C. Sequentially Mining Towards Search Space Reduction

However, since  $\mathcal{A}$  only knows when reception handler invocation is *imminent* (Section V-A), she must overestimate when the KEW begins and take stackshots intermittently to ensure some overlap, resulting in an unwieldy growing search space. Indeed, for cryptographic keys with identifiable traits (e.g., entropy, statistical, structural, and mathematical), identification would be trivial. However, this is generally *false*, especially for symmetric keys (Section II), and assumes that  $\mathcal{A}$  is aware of the type of key. Nonetheless, instead of resorting to an exhaustive search over the entire accumulated search space,  $\mathcal{A}$  continues to devise an *efficient SSR* algorithm (by Definition 3.1) to keep the remaining  $K$ -hammering effort within feasible levels. To  $\mathcal{A}$ ’s advantage, although keys can take many forms, they behave like other objects on the stack.

Since keyed cryptosystems contain excessive use of calculations (e.g., XOR operations) and inherently require that keys remain intact during use, some values on the cryptographic function’s stack frame will inevitably fluctuate, whereas the key will likely *remain constant*. Thus, if we would pour a set of consecutive stackshots into a funnel that filters infrequent values from frequent, we would essentially delimit areas likely to contain the key. To achieve such a funnel, we use pattern mining. Specifically, since stack frames are sequentially allocated and compartmentalized, we use Sequential Pattern Mining (SPM) [22] to exploit the sequential ordering property.

In SPM, a subsequence is a *sequential pattern* (Definition 5.1) if it appears frequently in a dataset  $D$ , and its frequency is no less than a *minimum support threshold* (*minsup*), i.e.,  $\geq \text{minsup}$  of stackshots must overlap with the KEW for keys to become frequent (appear as a candidate). Although several SPM approaches exist, many have the critical drawback of presenting too many patterns [22]. We opted for Maximal Sequential Patterns (MSP, Definition 5.2), which have also been used to find the frequent longest common subsequences to sentences in texts and to analyze DNA sequences [22]. MSP mining is appropriate since it presents: (i) a concise subset of *unique* patterns [23], which prevents running a VSW over the same data unnecessarily and (ii) allows us to constrain the number of permitted *gaps* (irregular words) between consecutive words in a pattern. Since keys must usually remain intact, we use (ii) to require that each consecutive word in a pattern also appears consecutively in a stackshot (i.e., *no gaps*).

The conjunction of both properties (i-ii) enables exploitation of the stack’s nature, where, within small time windows, some values remain constant (e.g., return addresses and *keys*) while others (e.g., temporaries in calculations) fluctuate. Note that since we use fluctuations to split the search space, the omission of values due to compiler optimizations (e.g., inlining and use of registers, see [15], [16]) can affect efficiency.

**Definition 5.1 (Sequential patterns):** A sequence dataset  $D$  is an unordered set of sequences:  $D = \{S_1, S_2, \dots, S_s\}$ , where each sequence  $S = \langle W_1, W_2, \dots, W_n \rangle$  corresponds to one *stackshot* and consists of an ordered list of words  $W_i$  (2 bytes in MSP430), where  $i$  denotes its index. A sequence  $S_a = \langle A_1, A_2, \dots, A_m \rangle$  is *contained* in another sequence  $S_b = \langle B_1, B_2, \dots, B_n \rangle$  if there exists integers  $1 \leq i < j < \dots < k \leq n$  such that  $A_1 = B_i, A_2 = B_j, \dots, A_m = B_k$ , and is denoted as  $S_a \sqsubseteq S_b$ . Here,  $S_b$  is a *super pattern* of  $S_a$ , while  $S_a$  is a *sub pattern* of  $S_b$ . A sequential pattern  $P$  is a sequence that is contained in one or more sequences in  $D$ .

**Definition 5.2 (Maximal sequential patterns):** A pattern  $P_a$  is said to be *closed* if there is no other pattern  $P_b$ , such that  $P_b$  is a *super pattern* of  $P_a$ ,  $P_a \sqsubseteq P_b$ , and their support is equal. A pattern  $P_a$  is said to be *maximal* if there is no other pattern  $P_b$ , such that  $P_b$  is a *super pattern* of  $P_a$ ,  $P_a \sqsubseteq P_b$  [23].

## VI. AN ARCHITECTURAL BLUEPRINT

### A. A High-Level Overview

Figure 4 depicts the consolidation of the conceptualized methodology (Section V). The attack commences by  $\mathcal{A}$  in-

jecting (or stitching together) and triggering some malcode, which, once activated, embeds system hooks to proactively secure its *timely* invocation (Section V-A). In the optimal case where it manages to attach callbacks directly onto the reception ISR (RX ISR), the malcode proceeds to remain stealthy until reception occurs. Upon reception, the reception interrupt flag (RXIFG, Step 1.1a) transitions, which causes the RX ISR to be invoked and call the malcode (Steps 1.2a and 1.3a). However, if hooking onto the RX ISR is too difficult, the malcode resorts to establish a periodic timer (T, Step 1.1b), where, on subsequent firings of the timer (Steps 1.2b through 1.4b), the malcode polls some pre-identified MMIO transceiver state (flag) (Section V-A) to determine whether reception is occurring (Step 1.5b). Regardless of the method, once the reception has occurred, the malcode establishes a system timer to take stackshots periodically (Step 2). On each firing of the timer (Steps 3 through 5), the malcode extracts a predefined range  $rg$  from where SP currently points and stores it within some unused memory region (Step 6) - e.g., the *heap*, since OS-support for dynamic memory allocation in MCUS is often lacking. When a sufficient number of stackshots have been accumulated, they are marshaled into packets and transmitted back to  $\mathcal{A}$  (Step 7), where  $\mathcal{A}$  applies the search space reduction (Step 8). Finally,  $\mathcal{A}$  concludes by hammering the remaining search space (e.g., by applying a VSW) for keys.

Note that  $\mathcal{A}$  must dissect the target system's underpinnings to set up the necessary callbacks and timers. However, due to space limitations, these details are found in Appendix A.

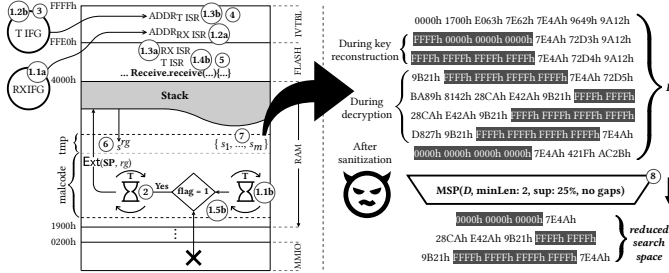


Fig. 4: Holistic work-flow of the key-acquisition attack.

### B. Modular Building Blocks

On a coarse-grained level, we separate the stages (Figure 4) into three versatile components. Two are interdependent and constitute the malcode – the *Watchdog* ( $\mathcal{WD}$ ) and the *Frame Extractor* ( $\mathcal{FE}$ ). The final component, the *Space Reductor* ( $\mathcal{SR}$ ), is the utilization of MSP data mining (Section V-C).

1) **Watchdog** ( $\mathcal{WD}$ ): The purpose of  $\mathcal{WD}$  is to commence a system timer to periodically invoke  $\mathcal{FE}$  when key exposure is imminent. Since we target the *software-oblivious* causality between reception and keyed cryptosystems, we can narrow the time window by periodically polling the transceiver's MMIO states (*observant* mode) or riding the RX-designated ISR (*dormant* mode). Although either approach effectively aligns the stackshots close to the KEW, dormant behavior is more stealthy since the CPU is free to enter (and stay uninterrupted) in low power modes (LPMs), but observer

behavior might be easier to accomplish. For the target system, we opted for a *hybrid* construct to maximize timeliness, where  $\mathcal{WD}$  rides the RX ISR and, upon invocation, starts a system timer to invoke  $\mathcal{FE}$  periodically. However, instead of taking stackshots immediately,  $\mathcal{FE}$  resists until a bit at a fixed memory location (which reflects whether transceiver communication is occurring) transitions, shifting stackshots closer to the reception handler's invocation (see Appendix A1).

2) **Frame Extractor** ( $\mathcal{FE}$ ): The  $\mathcal{FE}$  takes stackshots and transmits them to  $\mathcal{A}$  once it has accumulated a certain amount. Each stackshot comprises the region between the current SP and  $SP \pm rg$  bytes, depending on the direction of the stack.

3) **Space Reductor** ( $\mathcal{SR}$ ): Given stackshots  $D$ , MSP mining (Section V-C) is used to get a reduced search space (Figure 4).

## VII. EXPERIMENTS AND EVALUATION

We proceed to evaluate the  $\mathcal{SR}$ 's *efficiency* and *effectiveness*, where *efficiency* is quantified by the degree of the search space reduction and *effectiveness* by the ratio of keys in the initial (i.e., before applying  $\mathcal{SR}$ ) and reduced search spaces.

### A. Experimental Setup

We consider the reception handler in Fig. 2, where  $\mathcal{F}$  is substituted with two prominent open-source implementations of the Advanced Encryption Standard (AES) algorithm, namely: TinyAES [24] and one developed by Texas Instruments [25] (hereinafter referred to as TIAES). In short, the AES algorithm operates on 128-bit blocks, accepts key sizes of either 128, 192, or 256 bits, and comprises several layers that are applied to manipulate an input block in several successive rounds  $N_r$ , where  $N_r$  is a function of the key length (10, 12, or 14 rounds for 128-, 192-, or 256-bit keys). However, because TIAES works exclusively with 128-bit keys, we restrict both implementations to use 128-bit keys (AES-128). Nonetheless, for each round, AES derives (expands) a separate round key (subkey) from the initial key (master key) using its *key scheduling algorithm* (resulting in a total of  $N_r+1$  subkeys, where  $N_r=10$  for AES-128), which it supplies to the *key addition layer*. However, whether the entire key schedule (set of subkeys) is precomputed or subkeys are derived as needed is a design choice. For example, since TIAES is developed to accommodate memory scarcity, it reuses the same 16-byte memory area of the master key for holding subsequent subkeys during run-time. Contrarily, TinyAES targets energy scarcity and therefore precomputes the key schedule, enabling reusing the same schedule on subsequent executions. However, since the schedule in TinyAES is stored consecutive to the master key, it only increases the master key's size, and given its dominating size, necessitates a large  $rg$  for us also to capture surrounding values. Thus, for practical reasons, the schedule is kept *above the function level* (Section V-B), such that only the master key occurs on the stack. Therefore, for TinyAES, we assess  $\mathcal{SR}$ 's *effectiveness* by its ability to retain the master key in the reduced search space. For TIAES, however, we assess  $\mathcal{SR}$ 's *effectiveness* by its ability to retain *any* subkey (master key inclusive), which is justifiable since the remaining

schedule can be inferred (though the amount of key bytes required to infer the remaining schedule differs between AES key sizes). Table I shows each AES implementation’s cycle-accurate benchmarking results when executed within an emulated environment using MSPDebug v0.25 [26] in AES-128-ECB/decryption mode (including key expansion for TinyAES).

TABLE I: Cycle Count (CC) and Execution Time (ET, in ms) for decrypting one 128-bit block (CPU @ 4 MHz).

Impl.		O0	O1	O2	O3	O <sub>s</sub>
TinyAES	CC	390,279	34,408	28,986	23,612	33,234
	ET	97.57	8.602	7.247	5.903	8.309
TIAES	CC	52,256	16,642	13,411	8,505	15,747
	ET	13.064	4.16	3.353	2.126	3.937

To facilitate our experiments, we define a Capture Window (CW) as the *estimated* KEW and Capture Frequency (CF) as the number of stackshots per CW, which enables us to assess the stack at different temporal granularities by increasing and decreasing the CF. In general, CW must be large enough (over-approximated) or positioned close enough to overlap with  $\mathcal{K}$ ’s KEW, and CF be high enough to ensure sufficient  $\mathcal{K}$  captures ( $\geq$  the *minsup*, Section V-C) once the windows overlap.

Note, however, that the subkey lifespan is drastically different for TinyAES and TIAES. For TinyAES, the master key  $\mathcal{K}$  is kept intact, and its KEW will therefore be  $\approx$  the emulated ET (Table I). Thus, for a  $CW \leq ET$ , aligned perfectly with the actual KEW, we could confidently use any relative *minsup*  $\leq 100\%$  to reduce the search space without losing  $\mathcal{K}$ . However, for TIAES, each of the eleven subkeys will have KEW’s of  $\approx ET/11$  (the KEW of some subkeys will, however, be different since not all rounds are identical). Hence, in the same setup, where  $CW \leq ET$  is aligned with the beginning of decryption, then a relative *minsup*  $\leq 9.09\%$  should suffice to retain all subkeys in the reduced search space so long as  $CF \geq 11$ . Nonetheless, for experimental purposes, we let 25% and 6.25% (1/16) be our sufficiently permissive relative *minsup* thresholds for TinyAES and TIAES, i.e.,  $\mathcal{K}$  must appear in  $\geq 25\%$  of stackshots for TinyAES and  $\geq 6.25\%$  for TIAES.

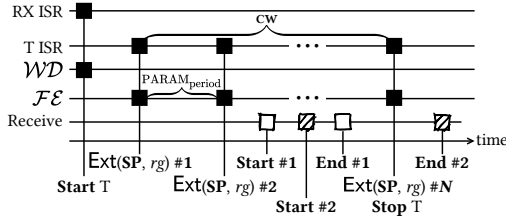


Fig. 5: Timeline ( $CF = N$ ) of how a CW might overlap the reception handler differently depending on when it begins.

Furthermore, given the high ET variation between implementations and optimization levels (see Table I), we consider a separate CW for each combination (i.e.,  $CW = ET$ ). Although these CWs can be considered optimal, note that  $ET \neq KEW$  and ET neglects delays until the reception handler is invoked (might take several  $\mu s$ ), the key reconstruction time, and

interrupt processing time – which increases slightly as CF increases. Thus, stackshots will inevitably occur at different offsets from the reception handler (see Fig. 5), and the CW will drift from the ET as CF increases. To reason about potential correlations between the CF and our experimental metrics – *efficiency* and *effectiveness* – we consider a set of CFs:  $\{4, 8, 16, 32, 64, 128\}$ . We use a common *rg* of 32 words (64 bytes) as the key is expected to occur relatively close to SP (see Section V-B), and a minimum pattern length of 2 words.

1) *Data Acquisition*: We devised two scripts: *getStacks*, for acquiring stackshots (using the malcode in Appendix A) from a Tmote Sky attached via an MSP430-JTAG (MSP-FET430UIF), and *spaceReducer* for MSP mining using seqwog v3.16 [27]. The *getStacks* script repeats  $n$  times (we set  $n = 15$ ), where it: for each combination of AES implementation, CF and optimization level, (i) erases the memory of the Tmote Sky, (ii) compiles and programs the reception handler with TinyOS v2.1.2 [4] on the Tmote Sky, (iii) extracts  $ADDR_{restore}$  (see Table II in Appendix A) from the handler’s assembly code, (iv) adjusts the malcode according to the CF and  $ADDR_{restore}$ , and writes it into memory – emulating *injection* (Section IV) –, (v) starts the Tmote Sky CPU, waits, and then stops the CPU, and (vi) finally reads the accumulated stackshots to a file – emulating exfiltration (Fig. 4). Note, to trigger the reception handler, we setup another Tmote Sky to transmit packets periodically. Also, to emulate code injection and commence the malcode, we added an inline assembly instruction at the tail of the reception handler, which branches to  $ADDR_{SE}$  at the *first* invocation of the handler. Once *getStacks* completed, *spaceReducer* was supplied with the 900 *independent* datasets, on which it: (i) preprocessed each dataset by pruning repetitions of “FF3Fh” from stackshot tails, since these words are read *past* the stack boundary (38FFh for the MSP430-F1611 MCU), and (ii) applied the MSP mining and yielded the reduced datasets.

## B. Empirical Results and Analysis

We proceed by considering Fig. 6, which presents details about the extracted datasets, for each CF, before and after MSP mining with relative *minsup* thresholds of 25% and 6.25% for TinyAES and TIAES, respectively (implementations separated by rows and bars colored by optimization level).

1) *The case of TinyAES*: Plot 6a illustrates how efficiently the search space is reduced in the case of TinyAES (average reduction of 79.95%). Note that while  $\mathcal{A}$  would run a VSW separately on each stackshot or MSP to find keys, Plot 6a considers the *concatenated* ( $\parallel$ ) search space, i.e., the concatenation of all stackshots in a dataset before and all MSPs after applying *SR*. Nonetheless, it is clear that even as we increase the CF significantly, the *reduced* search space, comprising the MSPs found in a dataset (batch of stackshots), remains considerably small (average size of 40.8 words). Furthermore, note that in the particular case when CF is 4, the reduction is expected to be negligible for a relative *minsup* of 25%, since the search space is only reduced when some patterns: (i) are infrequent, i.e., occur in  $< 25\%$  stackshots, which is not possible with



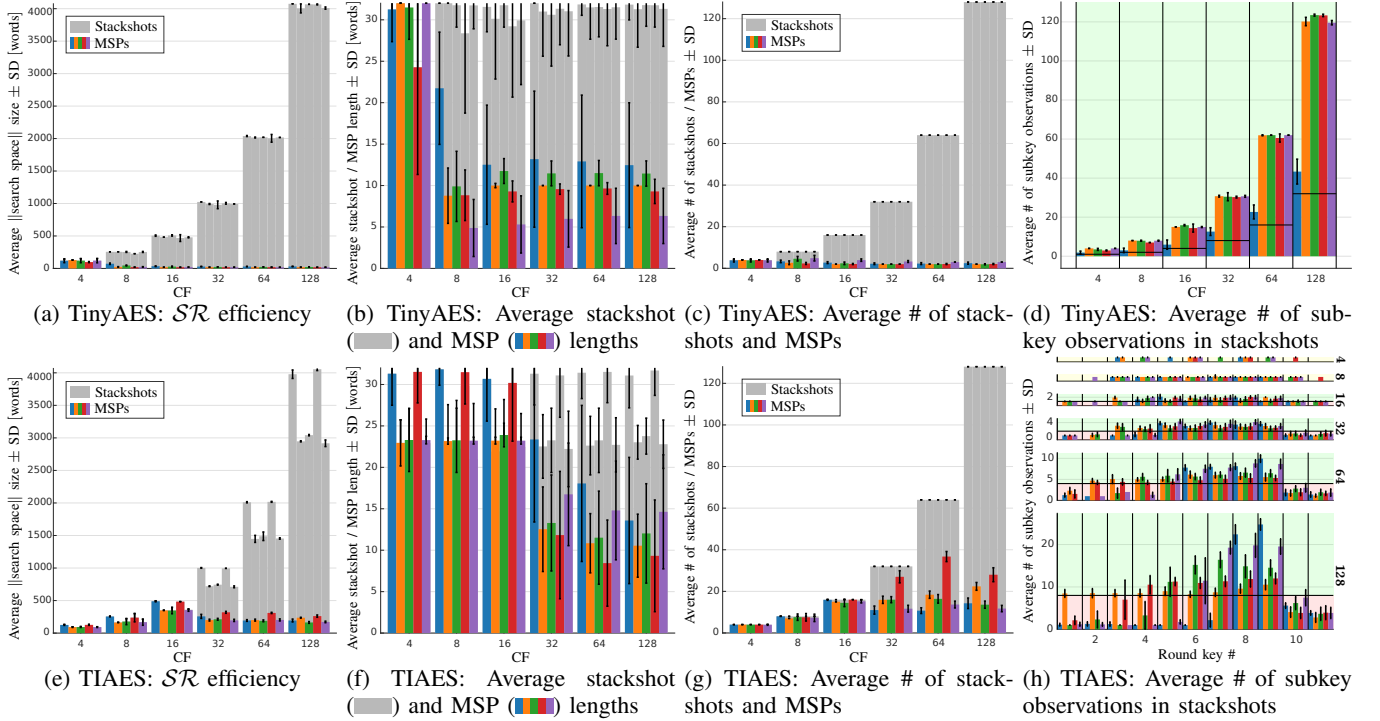


Fig. 6: Efficiency and effectiveness of the MSP data mining ( $\mathcal{SR}$ ). The figure shows means and Standard Deviations (SDs) of 15 independent datasets per combination of: AES implementation, CF and optimization level (00 ■; 01 ■; 02 ■; 03 ■; 0s ■).

four stackshots, or (ii) appear in multiple stackshots – as only the *super pattern* is presented (Section V-C). Excluding the insignificant CF of 4, the efficiency increases to 95.27%, and the average *reduced* search space decreases to 25.64 words. To better illustrate the effect of the data mining on a stackshot/MSP level, Plots 6b and 6c give further insights on the average stackshot/MSP lengths and count, respectively.

Regarding effectiveness (accuracy), Plot 6d illustrates how often the *entirety* of the master key appears on average in the datasets before the data mining, and the black horizontal lines denote the considered relative *minsup* threshold of 25%. We can see that the key almost always appears as *frequent* (above the threshold) and that for most optimization levels, we could confidently raise the *minsup* threshold without losing the key. Note that the key is observed less often for optimization level 00 because its final word lies on the edge of our *rg* of 32 words, i.e., many stackshots only include the partial key.

2) *The case of TIAES*: As with TinyAES, Plot 6e illustrates the reduction of the *concatenated* search space when using the relative *minsup* of 6.25% (average reduction of 42.87% and an average search space size of 228.31 words), and Plots 6f and 6g show the average stackshot/MSP lengths and count. As before, we expect negligible search space reduction for CFs {4, 8, 16}, since the support threshold comes into effect when the number of stackshots transcends 16. Excluding these CFs, we gain efficiency of 84.17% and a reduced search space size of 220.30 words. Note that although this search space is indeed much larger than for TinyAES, it contains *several* subkeys.

Regarding effectiveness, Plot 6h details how often the *entirety* of subkeys (the first subkey is the master key) appear on

average in the datasets before data mining, and the black horizontal lines denote the considered relative *minsup* threshold of 6.25%. Note that the latter half of the subkeys appear more frequently than others because when AES runs in decryption mode, it applies subkeys in reverse order (the master key *last*). Scrutinizing the results reveals that we seem to lose some subkeys at different optimization levels as we increase the CF. This trend occurs because our static CWs overlap less with the ET as we increase the CF due to the system spending more time processing interrupts (Section VII-A) – including the execution of some parts of the malware. Nonetheless, despite TIAES’s design approach making the key extraction more complicated – since the KEW of subkeys is *much* smaller than that of TinyAES’s master key (Section VII-A) –, the fact that some subkey always occurs above the horizontal line (the *minsup* threshold), indicates that *at least* one subkey always appears in the reduced search space, as part of an MSP.

## VIII. DISCUSSION AND POTENTIAL DEFENCES

We have demonstrated how an  $\mathcal{A}$  can systematically acquire *highly* ephemeral keys in MCUS. To make matters worse, for asymmetric cryptosystems, the KEP is even more extended, e.g., considering the F1611 CPU @ 8 MHz, 1024-bit RSA encryption takes hundreds of milliseconds and decryption several seconds [28]. Since we target keys *during* use, sanitizing keys *after* use is *insufficient*, and so is keeping keys *above* the *function level* as the key’s address on the stack is exploitable and challenging to avert (see Section V-B). However, since the attack’s effectiveness relies on our ability to approximate the use of keyed cryptography post-reception and incorporate

it as a trigger mechanism, any distortion of the approximation (e.g., delayed processing) mitigates the attack. Further, since carrying out the attack is difficult without disturbing program execution, Control-Flow Attestation/Integrity (CFA/I) [29] can be utilized to prohibit the attack. Nonetheless, such defensive solutions are *attack-specific* and do not directly resolve the KEP. To resolve the KEP, we must ensure that keys are either *useless when observed (captured)* or *unobservable*.

1) *Resolving the KEP*: We could decide never to store keys sequentially in memory. By (i) storing key bytes in different endianness, (ii) permuting the bytes' order, or (iii) scattering bytes throughout memory, we can directly affect  $\mathcal{A}$ 's ability to use the key. However, although (i) and (ii) prevent a naive VSW attack, they fail to prevent  $\mathcal{A}$  from plainly trying all byte (or word) permutations. The latter method (iii) ultimately *compels*  $\mathcal{A}$  to resort to an exhaustive attack since the search space might never contain all of the required pieces. However, realizing either method requires careful code instrumentation. Comparable strategies include white-boxing [30] and Moving Target Defense (MTD) [31]. With white-boxing, a given key is transformed into code that performs cryptographic operations without using the key material explicitly. However, embedment of keys into software makes key revocation difficult and is generally unsuitable in environments where keys must be frequently updated. With MTD, we make the key a moving target by rearranging its bytes regularly during run-time, which can be a viable mitigation tactic – assuming that the rearrangement scatters the bytes and not only permutes their order. Toward unobservability, we could confine keys to CPU registers [32]. However, achieving *secure* CPU-bound keys is presumptuous since it requires a sufficient number of *special* registers that are guaranteed to remain inaccessible to  $\mathcal{A}$ . More comprehensive solutions, e.g., Trusted Execution Environments (TEEs, e.g., ARM's TrustZone) and hardwareized CMs (e.g., Trusted Platform Modules), provide hardened interfaces for secure storage and use. However, being inherently resource-heavy, TEEs and CMs remain unavailable in most commodity MCUs.

## IX. CONCLUSIONS

There is a lack of adequate containment and trust regarding an embedded system's behavior, where sophisticated software attacks can circumvent standard key management techniques. By exploiting the strong causality between reception and keyed cryptosystems and the fact that cryptosystem implementations inherently require keys to remain exposed in memory during use, we have demonstrated how keys can be timely acquired off the memory stack during run-time. Our work serves a three-fold purpose: to reveal how the determinism of wireless-capable (event-driven) MCUS can pose an exploitable threat, study the effects of severe key-exposure attacks, and motivate the need for lightweight KEP-resilient protocols in resource-constrained MCUS.

## X. ACKNOWLEDGMENT

This work was supported by the European Commission, under the ASSURED project; Grant Agreement No. 952697.

## REFERENCES

- [1] N. Koutroumpouchos, C. Ntantogian, S.-A. Menesidou, K. Liang, P. Gouvas, C. Xenakis, and T. Giannetsos, "Secure edge computing with lightweight control-flow property-based attestation," *2019 IEEE Conference on Network Softwarization (NetSoft)*, pp. 84–92, 2019.
- [2] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μrai: Securing embedded systems with return address integrity," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*.
- [3] NVD, "CVE-2020-9395 Detail," 2020.
- [4] TinyOS, "TinyOS." [Online]. Available: [github.com/tinyos/tinyos-main](https://github.com/tinyos/tinyos-main)
- [5] R. Canetti *et al.*, "Exposure-resilient functions and all-or-nothing transforms," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2000, pp. 453–469.
- [6] B. Kaplan *et al.*, "Ram is key extracting disk encryption keys from volatile memory," 2007.
- [7] A. Shamir and N. Van Someren, "Playing "hide and seek" with stored keys," in *International conference on financial cryptography*, 1999.
- [8] J. A. Halderman *et al.*, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, 2009.
- [9] D. Genkin, L. Pachmanov, I. Pipman, A. Shamir, and E. Tromer, "Physical key extraction attacks on pcs," *Communications of the ACM*, vol. 59, no. 6, pp. 70–79, 2016.
- [10] B. Taubmann *et al.*, "Droidkex: Fast extraction of ephemeral tls keys from the memory of android apps," *Digital Investigation*, vol. 26, 2018.
- [11] P. Bajpai and R. Enbody, "Memory forensics against ransomware," in *Conference on Cyber Security and Protection of Digital Services*, 2020.
- [12] T. Klein, "All your private keys are belong to us," Tech. Rep., 2006.
- [13] T. Pettersson, "Cryptographic key recovery from linux memory dumps," *Chaos Communication Camp*, vol. 2007, 2007.
- [14] S. M. Hejazi *et al.*, "Extraction of forensically sensitive information from windows physical memory," *digital investigation*, vol. 6, 2009.
- [15] L. Simon, D. Chisnall, and R. Anderson, "What you get is what you c: Controlling side effects in mainstream c compilers," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018.
- [16] R. Chapman, "Sanitizing sensitive data: How to get it right (or at least less wrong...)," in *Ada-Europe International Conference on Reliable Software Technologies*. Springer, 2017, pp. 37–52.
- [17] C. Hargreaves and H. Chivers, "Recovery of encryption keys from memory using a linear scan," in *2008 Third International Conference on Availability, Reliability and Security*. IEEE, 2008, pp. 1369–1376.
- [18] T. Giannetsos, T. Dimitriou, I. Krontiris, and N. R. Prasad, "Arbitrary code injection through self-propagating worms in von neumann architecture devices," *The Computer Journal*, vol. 53, no. 10, 2010.
- [19] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.
- [20] M. Cooperation, "Ultra low power ieee 802.15.4 compliant wireless sensor module," *Datasheet*, 2006.
- [21] T. Instruments, "Msp430 embedded application binary interface," 2013.
- [22] P. Fournier-Viger *et al.*, "A survey of sequential pattern mining," *Data Science and Pattern Recognition*, vol. 1, no. 1, pp. 54–77, 2017.
- [23] —, "Vmsep: Efficient vertical mining of maximal sequential patterns," in *Canadian conference on artificial intelligence*. Springer, 2014.
- [24] kokke, "Tiny AES." [Online]. Available: [github.com/kokke/tiny-AES-c](https://github.com/kokke/tiny-AES-c)
- [25] T. Instruments, "AES-128." [Online]. Available: [ti.com/tool/AES-128](https://ti.com/tool/AES-128)
- [26] D. Beer, "mspdebug." [Online]. Available: [github.com/dlbeer/mspdebug](https://github.com/dlbeer/mspdebug)
- [27] C. Borgelt, "Seqwog." [Online]. Available: [borgelt.net/seqwog.html](https://borgelt.net/seqwog.html)
- [28] U. Gulen, A. Alkhodary, and S. Baktir, "Implementing rsa for wireless sensor nodes," *Sensors*, vol. 19, no. 13, p. 2864, 2019.
- [29] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.
- [30] S. Chow *et al.*, "White-box cryptography and an aes implementation," in *International Workshop on Selected Areas in Cryptography*, 2002.
- [31] J. Sianipar, M. Sukmana, and C. Meinel, "Moving sensitive data against live memory dumping, spectre and meltdown attacks," in *2018 26th International Conference on Systems Engineering (ICSEng)*, 2018.
- [32] T. Müller, F. C. Freiling, and A. Dewald, "Tresor runs encryption securely outside ram," in *USENIX Security Symposium*, vol. 17, 2011.
- [33] T. Instruments, "Msp430x1xx family user's guide (rev. f)," 2006.
- [34] —, "2.4 ghz ieee 802.15.4/zigbee-ready rf transceiver," 2011.



## A. Malcode Implementation Details

1) *Challenges*: Since ISRs are programmed together with program code into flash, we must reprogram the flash segment containing the timer and RX ISRs (Section VI-A) to achieve the necessary callbacks. Fortunately, MSP430 accommodates in-system programmable flash memory [33], as is typical for sensor motes to enable over-the-air field upgrades. Thus, with the advantageous Von Neumann memory model and lack of no-execute (NX),  $\mathcal{A}$  can easily reprogram flash from RAM.

a) *Administration of Periodic Invocation*: The MSP430-F1611 MCU has built-in MMIO configurations for two flexible 16-bit (asynchronous) timers, named Timer A (TA) and Timer B (TB), respectively. Each timer has multiple capture-compare units (CCUs), one control register,  $T_xCTL$ , for configuring the timer, e.g., specifying a mode of operation and clock source, and is coupled with one 16-bit counter register,  $T_xR$ , which increments or decrements (depending on the mode of operation) with each rising edge of the selected clock signal [33]. Each CCU has one capture/compare register (CCR), one capture/compare control register (CCTL) and operates in either capture or compare mode, as determined by the CCTL register. In compare mode, the value to be compared to is first loaded into the  $T_xCCR$ , and when  $T_xR$  equals that value, it raises the capture/compare interrupt flag (CCIFG) for that  $T_xCCR$ . Thus, to establish periodic interrupts, we configure TBCTL in *up* mode and say, TBCCR6, in compare mode, such that TBCCR6's CCIFG gets raised whenever TBR equals whichever value we store in TBCCR0 (the period). To react to this interrupt, we must reprogram the appropriate ISR, which for TBCCR6 is the IVTBL entry at FFF8h (used for TBCCR1 to TBCCR6). However, since multiple TBCCR CCIFGs are merged into this ISR, we must consult the timer interrupt vector register (TBIV) stored at 011Eh to determine whether TBCCR6 caused the interrupt. Finally, as the clock source (to regulate the period between stackshots), we use the auxiliary clock (ACLK), sourced from a 32 kHz watch crystal to achieve a granularity of 32 Cycles Per Millisecond (CPMS).

b) *Narrowing the Time Before the KEW*: The MoteIV Tmote Sky module [20] has a CC2420 [34] transceiver, which the MCU controls using an SPI link managed by UART0 and a series of I/O lines and interrupts. Specifically, MSP430 devices have up to 6 digital I/O ports, numbered P1 to P6. Each port has eight I/O pins, numbered Px0 to Px7, and four MMIO registers: PxSEL, PxDIR, PxOUT, and PxIN. When the CC2420 interrupts the CPU about an incoming packet, the packet is incrementally read from the CC2420 reception queue (RXFIFO) into the MCU's U0RXBUF memory buffer (located at 76h). On successive reads, the USART0 RX ISR (IVTBL entry FFF2h) [33] is invoked, and only when the entire packet resides in application memory then the reception handler gets invoked. Fortunately, the transceiver requires that the output of the Chip Select (CSn) pin, which is connected to pin 2 (3rd bit) of port 4 on the MCU, must remain low while there is any communication with it (read or write). Therefore, by polling

the CSn (bit 3 in the P4OUT register located at 01Dh) and noticing a transition from low (0) to high (1), then we are sure that the invocation of the reception handler is imminent.

2) *Crafting the malcode*: The malcode is a collection of five code segments and, for brevity, accepts 18 configurable parameters, which are presented together with arguments used during our experiments (Section VII) in Table II. Besides the predefined  $\mathcal{WD}$  (see Malcode 3) and  $\mathcal{FE}$  (Malcode 4), the malcode comprises a *Setup Engine* ( $\mathcal{SE}$ , Malcode 1) and an *ISR Injector* ( $\mathcal{ISRI}$ , Malcode 2), where  $\mathcal{SE}$  is the initial *triggering* of the malcode (Section VI-A). When invoked,  $\mathcal{SE}$  uses  $\mathcal{ISRI}$  to inject callbacks to  $\mathcal{WD}$  and  $\mathcal{FE}$  into the RX and timer ISRs. Note that for brevity, we omitted the final segment, which transmits stackshots to  $\mathcal{A}$ . Nonetheless, to maintain state and to reprogram flash, the malcode uses unoccupied RAM space, which, because MSP430 requires flash programming on a segment granularity, must be  $\geq 512$  bytes, such that  $\mathcal{ISRI}$  can temporarily copy, reprogram, and write back segments.

TABLE II: Malcode parameters and demonstrative arguments.

Parameter	Argument	Description
<b>Config</b>		
PARAM <sub>usartISR</sub>	FFF2h	Target USART0RX IVTBL entry
PARAM <sub>timerISR</sub>	FFF8h	Target TB IVTBL entry
PARAM <sub>CCTL</sub>	018Eh	Target TBCCTL6
PARAM <sub>CCIFG</sub>	0Ch	Value of TBCCR6 CCIFG in TBIV
PARAM <sub>runs</sub>	1	# of successive runs (receptions)
PARAM <sub>stackshots</sub>	CF	# of stackshots in each run
PARAM <sub>period</sub>	[CW/CF * CPMS]	Time between stackshots
PARAM <sub>rg</sub>	64	# of bytes to extract
<b>States (updated during run-time)</b>		
ADDR <sub>run</sub>	2000h	Current run count
ADDR <sub>stackshot</sub>	2001h	Current stackshot count
ADDR <sub>tmpPtr</sub>	2002h	Current offset in temporary storage
ADDR <sub>tmp</sub>	2200h	Temporary storage $\geq 512$ bytes
ADDR <sub>restore</sub>	*	Restoration memory address (Fig. 2)
<b>Demonstrative memory placement of the malcode</b>		
ADDR <sub><math>\mathcal{SE}</math></sub>	2004h	Start address of Malcode 1
ADDR <sub><math>\mathcal{ISRI}</math></sub>	202Ch	Start address of Malcode 2
ADDR <sub><math>\mathcal{WD}</math></sub>	20C4h	Start address of Malcode 3
ADDR <sub><math>\mathcal{FE}</math></sub>	20FCh	Start address of Malcode 4

a) *Methodical Execution*: Once the  $\mathcal{A}$  sends an activator packet which tricks the CPU's Program Counter (PC) register to point to the beginning of  $\mathcal{SE}$  (see Fig. 2),  $\mathcal{ISRI}$  is used to inject a callback to  $\mathcal{WD}$  in USART0's RX ISR and another to  $\mathcal{FE}$  in TB's ISR (Appendix A1). On each invocation,  $\mathcal{ISRI}$ : (i) identifies in which flash segment the target ISR is located, (ii) copies that segment into RAM (where it can manipulate it freely), (iii) overwrites the first two push statements in the ISR's prologue with a branch (jump) to the appropriate malcode component, (iv) clears the segment in flash memory, and finally (v) writes the manipulated segment back into its original slot. Thus, when either ISR triggers, the appropriate malcode is invoked. Subsequently,  $\mathcal{SE}$  gracefully resumes the reception handler by reverting control-flow to the reception handler's original return address (ADDR<sub>restore</sub>, see Table II).

At this stage, the malcode is armed but lies dormant as it awaits reception. Upon reception, the  $\mathcal{WD}$  awakens, and unless ADDR<sub>run</sub> has reached the PARAM<sub>runs</sub> threshold, TB is started to periodically (regulated with PARAM<sub>period</sub>) invoke  $\mathcal{FE}$ . Since different timers can run contemporaneously and

$\mathcal{FE}$ 's callback resides in TB's ISR's prologue,  $\mathcal{FE}$  consults TBIV to determine whether the CCIFG of the targeted CCU is raised (i.e., since we consider CCU6, whether TBIV has the value 0Ch [33]). If so,  $\mathcal{FE}$  determines whether the CSn has become high (Appendix A1), and if it has, advances to copy  $\text{PARAM}_{\text{rg}}$  bytes from where the SP currently points (excluding the first five words emitted by the interrupt and the  $\mathcal{FE}$ 's prologue) into  $\text{ADDR}_{\text{tmp}}$ , using  $\text{ADDR}_{\text{tmpPtr}}$  as an offset, and updates  $\text{ADDR}_{\text{stackshot}}$  and  $\text{ADDR}_{\text{tmpPtr}}$  accordingly. Once the  $\text{PARAM}_{\text{stackshots}}$  threshold is reached,  $\mathcal{FE}$  stops its timer, increments  $\text{ADDR}_{\text{run}}$ , and transmits the accumulated stackshots to  $\mathcal{A}$ . Finally,  $\mathcal{FE}$  resumes the TB's ISR.

#### Malcode 2: ISR Injector $\mathcal{ISRI}$ (152 bytes)

```

1  PUSH R2, R13 - R10
2  MOV #0x5A80, &0x0120 / stop WDT module
3  MOV.B R14, R13 / copy LSB as offset
4  SWPB R14 / swap MSB and LSB
5  MOV.B R14, R12 / MSB = segment start
6  MOV.B R12, R11 / copy MSB for testing
7  AND.B #0x01, R11 / 1 if MSB is odd
8  TST.B R11 / test if even or odd
9  JZ 0x6 / skip if even
10 DEC.B R12 / else, decrement MSB
11 ADD #0x0100, R13 / make offset odd
12 ADD #ADDRtmp, R13 / add segment offset
13 SWPB R12
14 MOV R12, R11 / copy segment start
15 ADD #0x0200, R11 / end = start + 512B
16 MOV #ADDRtmp, R10
17 MOV R12, R14 / copy segment start
18 MOV R14+, 0x0000(R10)
19 INCD R10
20 CMP R14, R11 / end of flash segment?
21 JNZ -0xA / go back 10 bytes
22 MOV #0x4030, 0x0000(R13) / swap(PUSH, BR)
23 MOV R15, 0x0002(R13) / swap(PUSH, callback)
24 MOV #0xA542, &0x012A / use MCLK/3
25 MOV #0xA502, &0x0128 / set ERASE bit
26 MOV #0xA500, &0x012C / remove LOCK bit
27 CLR 0x0000(R12) / erase segment
28 BIT #0x0008, &0x012C / check write status
29 JZ -0x6 / loop until done
30 MOV #ADDRtmp, R10
31 MOV #0xA540, &0x0128 / set WRT bit
32 MOV R10+, 0x0000(R12) / write word
33 INCD R12
34 BIT #0x0001, &0x012C / check busy status
35 JNZ -0x6 / loop until not busy
36 CMP R12, R11 / end of flash segment?
37 JNZ -0x10
38 MOV #0xA500, &0x0128 / remove WRT bit
39 MOV #0xA510, &0x012C / set LOCK bit
40 POP R10 - R13, R2
41 RET
```

#### Malcode 1: Setup Engine $\mathcal{SE}$ (40 bytes)

```

1  PUSH R15 - R14
2  MOV.B #0, &ADDRrun
3  MOV &PARAMusartISR, R14
4  MOV #ADDRWD, R15
5  CALL #ADDRISRI
6  MOV &PARAMtimerISR, R14
7  MOV #ADDRFE, R15
8  CALL #ADDRISRI
9  POP R14 - R15
10 BR #ADDRrestore
```

#### Malcode 3: Watchdog $\mathcal{WD}$ ( $\geq 54$ bytes)

```

1  PUSH R15 - R14
2  CMP.B #PARAMruns, &ADDRrun / done?
3  JZ line 12 / if yes, skip
4  BIT #0x0010, &PARAMCCTL / timer started?
5  JC line 12 / if yes, skip
6  MOV.B #0, &ADDRstackshot
7  MOV #0, &ADDRtmpPtr
8  MOV #PARAMperiod, &0x0192 / store in TBCCR0
9  MOV #0x0010, &PARAMCCTL / enable interrupts
10 CLR &0x0190 / reset TBR
11 MOV #0x1910, &0x0180 / TBCTL in up mode
12 MOV &PARAMusartISR, R15
13 ADD #0x0004, R15 / skip branch to self
14 BR R15 / allow ISR to progress
```

#### Malcode 4: Frame Extractor $\mathcal{FE}$ ( $\geq 88$ bytes)

```

1  PUSH R15 - R14 and R13
2  CMP.B #PARAMCCIFG, &0x011E / did the target timer fire?
3  JNZ line 24
4  BIT.B #0x04, &0x001D / CSn high?
5  JNC line 24
6  MOV #ADDRtmp, R13
7  MOV &ADDRtmpPtr, R13 / continue from tmpPtr offset
8  MOV R1, R14 / R1 is the SP
9  MOV R1, R15
10 ADD #0x000A, R14 / ignore 10 bytes (SR, PC, 3xPUSH)
11 ADD #0x000A, R15
12 ADD #PARAMrg, R15
13 MOV @R14+, 0x0000(R13)
14 INCD R13
15 CMP R14, R15
16 JNZ -0xA
17 INC.B &ADDRstackshot
18 ADD #PARAMrg, &ADDRtmpPtr / increment offset
19 CLR &0x0190 / reset TBR
20 CMP.B #PARAMstackshots, &ADDRstackshot / done?
21 JNZ 0x8
22 MOV #0, &PARAMCCTL / disable interrupts
23 INC.B &ADDRrun

CC2420 TRANSMIT


24 MOV &PARAMtimerISR, R15
25 ADD #0x0004, R15 / skip branch to self
26 POP R13
27 BR R15 / allow ISR to progress
```