Safe and Secure?
On the Timing Analysability of Cryptographic Implementations

Alexander Stegmeier*, Peter Knauer†, Philipp Schubaur†, Christian Piatka*,
Dominik Merli† and Sebastian Altmeyer*
*University of Augsburg, Department of Computer Science, Augsburg, Germany
Email: [lastname]@es-augsburg.de
†Augsburg Technical University of Applied Sciences, Institute for Innovative Safety and Security, Augsburg, Germany
Email: [firstname].[lastname]@hs-augsburg.de

Abstract—Hard real-time systems are increasingly vulnerable to cyberattacks. Since real-time systems represent a significant proportion of safety-critical systems not only established safety standards but also security standards have to be considered. In particular, standard cryptographic libraries are required to reach an adequate level of protection.

In this study, we investigate whether it is possible to ensure security and hard real-time without compromising either side. Thus, we examine relevant state-of-the-art cryptographic primitives provided by one of the de-facto standard libraries Mbed TLS, which is widely-used in embedded systems. We investigate the possibility to derive a Worst-Case Execution Time (WCET) for these primitives and review the code base with regard to compliance on safety-related coding guidelines. In addition, we assess the relevant aspects when security concerns must be considered in the safety-related context. Our research reveals several obstacles to fully apply Mbed TLS in hard real-time systems.

Index Terms—Security, Safety, Real-Time, Cryptography, Static timing analysis

I. INTRODUCTION

According to established safety standards, like IEC 61508 [1] or DO-178B [2], safety-critical systems have to undergo a certification process to ensure a specific level of safety by avoiding erroneous system behaviour. Hard real-time systems represent a significant proportion of the safety-critical sector [3]. These systems must satisfy timing requirements by providing results within predetermined timing intervals. The verification process addresses the timing requirements of safety-critical systems. Nevertheless, the interconnection of numerous systems and sensors has created new opportunities for attacks, thereby increasing the vulnerability of safety-critical systems to cyberattacks that intend to steal or manipulate the information being processed. The IEC 61508 standard stresses the significance of security in safety-critical systems by specifying essential safety requirements. It recommends IEC 62443 [4] and other standards as key references for in-depth information on this subject.

The IEC 62443 standard specifies the requirements for a secure system, with the goal of attaining adequate levels of Confidentiality, Integrity, Availability (CIA) of processed information [4]. Utilizing cryptographic algorithms provides protection against breaching these principles by cryptographic principles. It is highly recommended to exclusively use validated cryptographic algorithms due to the intricacy of developing such algorithms. Due to the potential of unnoticed errors introducing security flaws, it is not recommended developing custom cryptographic implementations. This is in accordance with the commonly accepted maxim within the security industry of "don't roll your own crypto" [5].

This research examines the conflict of interest at the edge of security and real-time requirements, seeking to align the demands of hard real-time systems with security concerns. Consequently, we (experts in safety and security) address the research question:

Can we ensure security and safety via real-time without compromising either side?

This question requires clarification regarding whether cryptographic algorithms and their implementations meet the implementation requirements to be able to pass the certification process for application in hard real-time systems. Furthermore, there is a need to assess the extent to which we have to accept compromises and whether we can still reasonably satisfy the requirements of both fields after doing so.

To answer our research question, we take on the role of a developer tasked with designing a system that addresses security concerns while successfully passing the certification process of common safety standards. Thereby, we narrow our focus to the requirements on the implementation of the system. To ensure that we develop a safe and secure embedded system, we employ common cryptographic algorithms provided by a widely used library in the embedded domain. Furthermore, we consider the use of a suitable hardware platform given the embedded nature of our system.

We focus on symmetric cryptography and authentication mechanisms as they could be particularly relevant for safety-critical embedded systems. We emphasize utilizing various methods as building blocks for the algorithms being explored

to represent a broad range of cryptographic methods. This study examines two modern cryptographic algorithms, Advanced Encryption Standard (AES) for symmetric encryption and Hash-based Message Authentication Code (HMAC) for data authentication, which are relevant to hard real-time systems. As Mbed TLS [6] is a quasi-standard library for cryptographic algorithms in embedded systems, we utilized the AES and a HMAC implementations provided therein. We selected the AES algorithm in Cipher Block Chaining (CBC) mode for both encryption and decryption functions, which is acknowledged as a typical mechanism. On the other hand, we focused on HMAC to guarantee message integrity by using a Message Authentication Code (MAC) based on hash calculations. We use the Infineon XMC4500 microcontroller [7] as our reference hardware platform.

A significant challenge in achieving safety certification on the highest integrity levels is performing a *static timing analysis* to establish dependable upper limits on the software's Worst-Case Execution Time (WCET). Moreover, it is critical to avoid excessive overestimation of the obtained results as such timing bounds may render the application infeasible for real-world scenarios. Notably, the demand for a low overestimation restricts code design and prohibits structures that cannot be analysed effectively, such as dynamic memory allocation [8].

Our study explores the feasibility of analysing cryptographic algorithms through static timing analysis for use in embedded systems with a cutting-edge timing analysis tool. We use the aiT tool from AbsInt [9] to conduct our analysis since it supports the timing verification requirements of various safety standards (e.g., ISO 26262 [10] and DO-178B [2]). We report the challenges encountered during the process, as well as the findings obtained. Furthermore, there are several guidelines available that support software development according to previously mentioned safety and security standards. Hence, we also review the code base according to the compliance with those guidelines.

Our investigation has revealed existing issues and potential causes of overestimation, which limit the accuracy of timing upper bounds provided by the analysis. Furthermore, we propose solutions to circumvent problematic code structures when using the library under investigation alongside a recommendation to adjust its implementation to mitigate safety concerns.

The remainder of this paper is structured as follows: Section II reviews related work. Afterwards, Section III discusses concerns in development affecting safety aspects and Section IV refers to security aspects. Section V presents the procedure how we conducted the static WCET analysis. It follows the investigation of Mbed TLS in Section VI, including a description of the necessary analysis efforts and gained results. A discussion about revealed problems and recommendations to simultaneously satisfy real-time and security is presented in Section VII. Finally, Section VIII concludes this study.

II. RELATED WORK

Combining of security and real-time application requirements were introduced by Mohan in 2008 [11]. The publication addresses the associated issues but does not provide an in-depth analysis.

Research in the crossover field of security and cryptography, and real-time safety is a current area of investigation. Mueller studied security mechanisms for prevention, detection and recovery, resilience and deterrence of attacks [12]. A 2022 publication by Ke [13] presents a method for assessing the feasibility of ciphers in embedded Real-Time Operating System (RTOS) through timed automata, providing a detailed examination of AES implementation properties. There exists a real-time related benchmark suite, namely TACLe bench [14]. This suite is intended to extensively test timing analysis tools and contains some security primitives as part of its workload. In contrast to the libraries considered in this study, the implementations in TACLe are not validated according to security concerns.

Yarza proposed a framework in the intersection of safety and security in 2022 [15]. In comparison, the present paper evaluates the WCET of a given system on a scientific basis. Trilla conducted an analysis of the predictability of time and side-channel attacks in cache memory. The study demonstrated that whilst randomization methods enhance time-predictability, they do not offer protection against side-channel attacks and vice versa [16]. The detection of timing violations can be solved by a performance counter, as introduced by Carelli in 2019 [17]. Furthermore, he explains a possible side-channel attack vector based on the ability to read the performance counter as well as the associated mitigation strategy. Zimmer has developed techniques for identifying intrusions in an application by means of self-checks via the scheduler of an operating system [18]. In the same field of research, McDonald conducted a study in 2022 using timing anomalies to detect intrusions in real-time systems via a dedicated kernel module [19]; in contrast, modifications of the kernel is not in scope of this paper.

Blackham ran a WCET analysis for *seL4*, a micro kernel [20] whereas this study examines cryptographic primitives. Horga published his doctoral thesis, evaluating the performance via a measurement-based WCET analysis and security of GPU-based applications [21]. In contrast, the presented research focuses on resource-restricted embedded devices. Völp discovered tasks that are usually analysed in regard to accidental faults experience fundamentally diverse outcomes when impaired by malicious attacks [22].

This paper examines the Mbed TLS library as one of the de-facto standards for embedded devices. For this reason, no comparison of cryptographic algorithms has been made as in the publications by Wollinger [23] or Manifavas [24]. Furthermore, no security analysis of different cryptographic libraries, as presented by Silde [25], was done. Additionally, a resource evaluation for energy consumption, as explored by Kerkhof [26], was not within the project's scope.

III. SAFETY CONCERNS IN DEVELOPMENT

We aim to develop a real-time system that is certifiable according to common safety and security standards. According to safety requirements the software must provide a sound upper bound on its execution time. *Static timing analysis* is able to provide such bounds. This analysis method relies on static methods to determine possible execution paths from assembly code and calculate their execution time bound. A successful timing analysis is mainly influenced by three different aspects. These are the structure of the analysed software, the hardware used for execution and the capabilities of the applied analysis tool.

A. Guidelines for safety-critical software

Various sources in literature provide information about characteristics of software that affect its analysability. Several available guidelines facilitate the development of safety-critical code. Different authorities recommend their application. Here, we shortly present MISRA-C [27] and "The Power of 10" [28]. Both guidelines are well known in safety industry and are referred to in other safety coding standards like the JPL Institutional Coding Standard for the C Programming Language [29]. Additionally, Gebhard et al. [8] focuses on the implications of such guidelines for conducting a static real-time analysis. The investigated rules that influences a software's real-time analysability typically target one or more of three aspects:

The first aspect concerns the reduction of infeasible paths in code or a simplified automated detection of such paths. MISRA-C targets this aspect for example by requiring that there shall be no unreachable code [27]. Causing a similar outcome, "The Power of 10" strongly restricts the usage of pointers. Specifically, amongst others it allows only one level of dereferencing. Furthermore, function pointers are completely forbidden, unless there is a "very strong justification" for their usage [28].

The second aspect ensures a code structure that enables the analysis tools to automatically determine the maximum number of executed iterations of loops (known as loop bounds). For example, rules of MISRA-C forbid any objects of floating type in the controlling expression of `for` statements or that numeric variables being used for iteration counting of `for` loops shall not be modified in the loop body [27]. "The Power of 10" is even stronger in requiring that for each loop it shall be trivial for a checking tool to statically prove that the loop cannot exceed a preset upper bound on the number of iterations [28].

Finally, the third and last aspect is the reduction of memory accesses that cannot be resolved to a precise address (imprecise or unknown memory accesses). Targeting this aspect, MISRA-C requires that dynamic heap memory allocation shall not be used [27]. In "The Power of 10" the same rule is stated in a slightly relaxed fashion. It forbids the usage of dynamic memory allocation after initialization [28]. Note that the rule targets the initialization phase of the entire system or application and not the initialization of a single cryptographic procedure at runtime.

The guidelines facilitate a static execution time analysis [8]. Hence, they reduce the need for developers to provide additional information to conduct a successful analysis. The reduction of that effort is a relevant criterion in safety-critical software. Identifying relevant points in code and gathering suitable information about the execution characteristics can be challenging and is associated with additional effort. Moreover, manually provided information about the execution behaviour weakens the soundness of the analysis, since this information is not covered by the guarantees provided by a previously verified analysis tool.

B. WCET Analysis Tool

We apply aiT by AbsInt [9] to investigate the real-time capabilities of the code we are examining. It is a state-of-the-art tool for providing static timing analyses on executables. Thereby, it disassembles the binary file and performs the analysis via abstract interpretation [30] on assembly instruction level and a cycle accurate model of the hardware. However, if there are sections in disassembly code that cannot be analysed statically, manual effort is needed to provide hints about the execution behaviour. For example, one may need to provide the maximum number of iterations a given loop may perform.

According to Wilhelm et al. [31], the analysis comprises several analysis stages. The binary is analysed to extract the control-flow graph. Additionally, a loop and value analysis is performed. Thereby, the control-flow graph is constructed based on basic blocks, i.e., pieces of code that contain exactly one entry and one exit point. The above analyses seek to automatically include ceilings for the number of iterations of each loop and intervals in which the values of variables can be determined.

The pipeline analysis and the path analysis follow. The pipeline analysis determines the WCETs of basic blocks when executed on the target hardware. This analysis implicitly includes a cache analysis in order to be able to determine cache hits and cache misses. The path analysis uses all previous analyses and determines the worst possible execution path of the analysed program. Thereby, it uses the control flow graph in conjunction with the possible variable values and loop limits in order to determine the possible paths. The longest execution path is specified by combining the determined execution paths and the execution times of the basic blocks. The total execution time of the longest path is the program's WCET and thus the final result of the WCET analysis.

C. Hardware Platform

Many current microcontrollers utilize hardware structures that complicate a precise static WCET analysis. Hence, the applied analysis tool must support the hardware applied for our system design.

Since the Infineon XMC4500 microcontroller [7] is supported by our analysis tool [33], it is a suitable representative for executing hard real-time applications where timely and accurate responses are critical. It is a member of the Infineon XMC family and designed for industrial applications such as
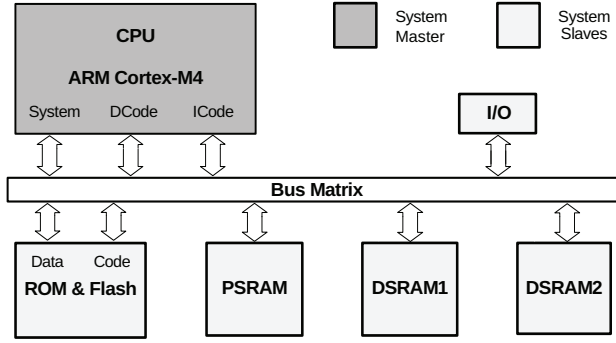
Fig. 1: The Infineon XMC4500 microcontroller, based on [32]

electric motor control, industrial connectivity and sense & control applications [7]. Figure 1 shows a simplified block diagram of the microcontroller. It comprises an ARM Cortex-M4 core, a boot ROM module, a flash module with 4 KB instruction cache and SRAM modules with sections for program, communication, and data memory. While SRAM memory can be accessed within one cycle, the flash memory exhibits an access latency of 22 ns [32]. Given the operation frequency of 120 MHz, there is a latency of 3 cycles for each access to the flash memory that does not hit a cache entry.

This study uses the XMC4500 Relax Lite Kit evaluation board [34] to execute the investigated cryptographic implementations. Thereby, the experiments performed on that board serve to verify the correct functional behaviour of our implementations and to gather measured execution times. Although we conduct our experiments on this particular architecture, we assume our study to be valid beyond it.

IV. SECURITY CONCERNS IN DEVELOPMENT

To ensure software security in development, it is crucial to follow guidelines for constructing a secure software system that is engineered from the ground up with security in mind.

A. Guidelines for security-aware software

Cryptographic algorithms are essential to ensure the integrity and confidentiality of data in embedded systems. Given their complexity, their deployment requires a professional approach to mitigate vulnerabilities. Cryptographic libraries, which have been developed and maintained by expert communities, undergo rigorous testing to ensure reliability. These libraries address common concerns such as memory safety and buffer overflows. It is recommended to employ established and verified cryptographic libraries approved by the security community. Such libraries are optimized for achieving low-overhead security mechanisms, while ensuring that system functionality remains uncompromised. Foundational selection criteria for cryptographic libraries in embedded systems should prioritize protection goals like CIA. Integrity is especially essential for safety-critical applications since it guarantees the dependability and accuracy of processed information [35].

Utilizing dependable cryptographic libraries bolsters the security posture of embedded systems, shielding them from cyber threats and vulnerabilities.

B. Cryptographic Libraries for Embedded Systems

The selection of cryptographic libraries for embedded systems is complex due to the constrained availability and specific requirements of these platforms. It is essential to choose libraries that are widely recognized as well as thoroughly tested but also align with the security paradigm of using only validated implementations. In the following section, we provide details on noteworthy libraries that meet the unique demands of embedded cryptography:

Mbed TLS [36]*:* Mbed TLS is an open-source library that is optimized for use in embedded systems. It provides a wide range of cryptographic primitives and can be customized for different applications by disabling unnecessary modules. This allows developers to minimize the overall memory footprint, making it suitable for resource-constrained environments. Furthermore, its permissive licensing model and robust community support enhance its accessibility for both developers and researchers.

OpenSSL [37]*:* As a highly regarded cryptographic library, OpenSSL fully supports both SSL and TLS protocols. While OpenSSL's broad range of cryptographic tools and algorithms has led to widespread usage in various systems, its complexity may render it unsuitable for resource-constrained embedded systems. Additionally, due to its extensive complexity, OpenSSL may prove too resource-intensive for the specific safety-critical systems included in our study. As a result, Mbed TLS may be a more optimal choice than OpenSSL.

wolfSSL/wolfCrypt [38]*:* WolfSSL is an SSL/TLS library designed for embedded systems, and IoT devices. It is backed by wolfCrypt, an underlying cryptographic library that can be used independently or as part of wolfSSL. The library is focused on achieving high performance, compactness, and compliance with industry standards. Its modular design allows users to select only the necessary features. However, due to its wider recognition, easier licensing, integration with major operating systems like Zephyr, and comprehensive offerings, Mbed TLS is a better-suited candidate for our research.

TinyCrypt [39]*:* TinyCrypt is a minimalist cryptographic library that provides fundamental cryptographic primitives, making it an ideal choice for devices with limited memory and computational resources. Although TinyCrypt's minimalistic design is advantageous for certain embedded applications, it lacks the extensive cryptographic capabilities required for our in-depth research, as opposed to Mbed TLS.

GnuTLS [40]*:* This open-source library enables secure communication protocols, such as SSL, TLS, and DTLS. GnuTLS takes a flexible and performance-oriented approach to cryptography with a modular architecture that facilitates selective feature inclusions. Despite its flexibility and support for secure communication protocols, GnuTLS is not tailored to embedded systems like Mbed TLS, which is a better fit for our research focus.

TABLE I: Overview comparison, based on GitHub stars, recency, license and binary size

	GitHub Stars [42]	Last Release	Licensing Model	Binary Size [43]
Mbed TLS	4.8k	Jan '24	Apache 2.0	664 KiB
OpenSSL	23.8k	Jan '24	Apache 2.0	5,000 KiB
wolfSSL	2.1k	Dec '23	GPL-2.0	1,768 KiB
TinyCrypt	0.4k	Aug '17	3-Clause BSD	–
GnuTLS	0.4k	Jan '24	LGPL 2.1	–
LibTomCrypt	1.5k	Jul '18	Unlicense	–

LibTomCrypt [41]*:* LibTomCrypt is a versatile cryptographic toolbox that caters to a variety of applications, from embedded systems to server applications. Its modular design enables users to select required components. However, challenges related to infrequent updates and licensing concerns have been reported. Thus, Mbed TLS is a more reliable option for our study due to its consistent updates and clear licensing.

The listed libraries differ in terms of distribution, code quality, licensing, and implementation standards, according to table I. Our evaluation, coupled with Stancu et al.'s [44] research, indicates that Mbed TLS is the most favorable option for conducting our analysis in real-time environments. Its design is customized specifically for embedded systems, which is in line with the context of our study. The library's emphasis on modularity ensures that only essential features are included, reducing potential vulnerabilities and preserving vital resources, which is essential for security-critical applications. In addition, its extensive range of cryptographic functions ensures resilient and secure communication channels, which are essential for real-time systems. Furthermore, the integration of Mbed TLS with popular real-time operating systems, such as Zephyr OS, not only confirms its capabilities in real-time applications, but also establishes its reliability. With its adaptability to various applications and recognition as a de facto standard for embedded systems, Mbed TLS stands as an optimal candidate for in-depth analysis in real-time security-critical scenarios. We have therefore decided to focus our efforts on Mbed TLS in release v3.2.1 and leave the extension of our findings to multiple libraries to future work.

V. CONDUCTING THE WCET ANALYSIS

We conduct a WCET analysis by aiT from AbsInt by first performing an analysis run followed by a phase of investing manual effort based on the information gathered from the previous analysis run. Thereby, the manual effort provides information to the tool via annotation statements in order to eliminate situations that impede the analysis until an aspired level of precision is reached.

A. aiT Configuration Validation

Before starting the analysis, we ensure that our tool configuration is sound. Hence, we analyse small synthetic benchmarks which validate our configurations first.

We validate if aiT works correctly for the applied hardware. Thus, we perform some simple experiments and check the correctness of the analysis runs and if they exhibit the expected results. The experiments aim to reduce the influence of software structures like conditional execution. They execute a number of nop instructions (10, 100 and 1000) within the body of a for loop iterating for a fixed number of times (10 and 20).

Since the nop benchmarks include only a small number of loops, routines or conditional execution, the number of basic blocks is small, too. According to these characteristics the results of the static analysis should be close to the measured values but still higher than or equal to them. Our tests reveal that the nop benchmarks show the expected behaviour. These results give us confidence that we have the tool configured correctly.

B. Problem Types

As introduced in Section III-A, safety-related literature refers to three aspects to facilitate static real-time analysis. These aspects are:

- Reduction of infeasible or unresolved execution paths
- Limitation of the maximum number of loop iterations
- Reduction of imprecise or unknown memory accesses

Our experiences gained during the analyses for this study strengthen the finding that violating these aspects are the major cause for an increased effort to reach a certain level of precision of the analysis result.

The analysis may encounter various types of problems related to the mentioned aspects. One type of problem prevents the analysis to find possible execution path. An example is the usage of dynamically assigned function pointers. Such pointers prevent a static decision about possible execution paths. Another problem is conditional executions, which contain paths that are never executed. Those infeasible paths may be assumed to be on the critical path and thus, may increase the analysed WCET upper bound significantly. Additionally, there may exist loops which are not bounded automatically. If the number of executed iterations of a loop cannot be bounded or is unknown, it is not possible to derive a safe upper bound since it is theoretically possible to spin on that loop forever. Finally, unknown or imprecise memory accesses may cause overestimation since the access delay of the slowest possible memory must be assumed. Furthermore, if caches are used, each such access destroys the current state of the cache analysis, resulting in overall poor cache performance in the analysis results.

C. Solving Problems

We solve the presented problems by providing additional information via annotations. Hence, the provided annotations are not covered by the verified correctness of the tool and the responsibility for their correctness remains to the developer.

The impacts of the mentioned problems to the analysis lead to the following solving order to conduct the analysis efficiently.

1) Ensure recognition of all possible execution paths
2) Exclude all infeasible paths
3) Determine maximal number of executions for each path
4) Refine unknown or imprecise memory accesses

First, we identify every possibly taken execution paths to ensure that no required annotations will be missed in subsequent analysis steps. Typically, this is done by solving all issues due to unresolved paths. They originate e.g., at calls to dynamically set function pointers. Thereby, the provided information target the analysis stage of extracting the control flow. If such code structures are encountered, the tool will issue warnings. After this step, the analysis is able to calculate the maximum reachable call graph.

Subsequently, in the second step it is reasonable to exclude all infeasible paths to restrict the invested effort of subsequent steps only to relevant paths. This step can only be solved via manual code or call graph review and mainly consists of an investigation of all conditional executions in order to find execution paths that are actually impossible to enter. This step facilitates the path analysis and restricts the number of possibly executed paths either directly or by restricting variable values that affect control flow. Depending on the size of the code base this may cause a large amount of manual effort and can be a challenging task. Moreover, this task has to be considered error-prone since it cannot be supported with automatically calculated information from the analysis. After the first two steps the set of all possibly executed paths is known, and it is the minimal set of paths.

After fixing the call graph the next step decides the number of executions for each path. Typically, this issue arises for loops. The tool is partly able to decide the maximum number of iterations automatically. However, this ability depends on the code structure and exit conditions of loops. In case a loop cannot be bounded the tool throws a warning. Additional effort may arise if bounds are dependent on the calling context. If the relevant information for the loop bound is determined in higher levels of the calling hierarchy, further annotations may be necessary for each call level. The information provided in this step are utilized by the loop and value analysis of the tool.

Finally, the last step refines the results by stating target addresses of memory accesses. Providing information about memory access locations affect the pipeline analysis and its implicitly conducted cache analysis in terms of influencing the calculated WCET bounds for the basic blocks. The tool classifies each possible access as *exact*, *nearly exact*, *imprecise* and *unknown*. Thereby, accesses that are located within a maximum address range of 1024 bytes are classified as *nearly exact*. Otherwise, if the addresses of each of the access's invocation can be specified, but the range is wider than stated above they are classified as imprecise. If at least one of the invocations of the memory operation cannot be decided statically it is classified as *unknown*.

The refinement of memory access targets may be conducted with various extents of precision. One option is to examine the accesses as detailed as possible and annotate the smallest possible set of target addresses. While this approach leads to the most precise upper bounds, it may cause a tremendous amount of analysis effort including increased risk for false annotations. The other possibility is to only state address ranges that restrict each access to the targeted memory module. Such imprecise ranges destroy cache analysis but enable the consideration of the correct memory access latencies. Hence, the trade-off is to accept a larger overestimation to reduce the analysis effort.

VI. INVESTIGATION OF MBED TLS

We put ourselves in a position of a developer tasked with designing a system that addresses security and safety concerns. Hence, we focus on a set of representative algorithms that are particularly relevant and most likely chosen for embedded systems. Our objective is to evaluate the principle possibility of simultaneously serving the fields of security and real-time without compromising either side. Hence, we avoid additional complexity such as compiler optimization in the analysis of our study. Therefore, if not stated otherwise, during this study, we focus on binary files compiled with optimization level -O0, since higher levels tend to complicate a precise analysis.

A. Implementations under analysis

We chose AES encryption for symmetric cryptography. The CBC mode is utilized as the basic version of feedback from the cryptographic result into the input. Additionally, we take a closer look at the HMAC implementation of Mbed TLS as representative of authentication mechanisms. In contrast to AES, this mechanism is based on hash calculations instead of block cipher cryptography. SHA-256 is a secure hash algorithm, based on the Merkle-Damgård construction [45].

Mbed TLS provides the ability to statically exclude the code of unused hash methods at design time via macro definitions. This option allows resource efficient compilation by significantly reducing the binary size. We investigate the impact of such optimizations on timing analysis as well. Thus, in this study we explore two variants of HMAC, one statically configured variant and one without this optimization. Subsequently, we denote the resource efficient variant of HMAC as *static config*. Defined by the chosen SHA-256 input hash, the HMAC computes a 32 byte tag with an associated 64 byte key while operating on 64 byte blocks. We considered Cipher Block Chaining Message Authentication Code (CMAC) for the investigation of an authentication algorithm, too. However, this algorithm is based on block ciphers and thus, uses the same technique as AES. Therefore, we decided against a detailed analysis of CMAC in favour of HMAC. Nevertheless, we show some problems we would face when conducting an analysis of CMAC, since they are not present in AES or HMAC.

Figure 2 shows the characteristics of the implementations under investigation in terms of code size and control flow structures. The graph provides insight about five attributes of the software under analysis. Two of them target the size of the code. These are the number of instructions in code (instructions) and the size of the binary code of bytes (bytes). The remaining three aspects provide information about the
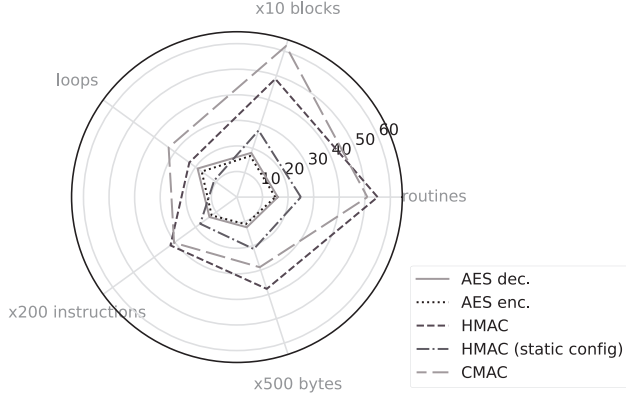
Fig. 2: Code characteristics of analysed Mbed TLS implementations

code structure. These are the number of loops included in each primitive (loops) and the number of routines in code (routines). Finally, the third aspect about code structure states the number of basic blocks of the programs (blocks).

The depicted characteristics confirm that the HMAC implementation with statically excluded code for unused hash methods is significantly smaller than its counterpart in terms of all presented characteristic (including code size). Looking at symmetric cryptography implementations, the code base of AES encryption and decryption show very similar characteristics. This is an expected observation since the performed operations in both methods are closely related. A comparison of AES-CBC and HMAC reveals that HMAC is larger in terms of code size and includes significantly more basic blocks and routines. However, the number of loops is similar. The CMAC implementation exhibits a code size which lies between both variants of HMAC. However, the number of routines are similar to HMAC without static configuration and in terms of loop number and basic blocks it exceeds all presented implementations.

Basically, all investigated implementations are executed in three phases, which are initialization, processing, and finalization. During initialization the execution context is allocated and prepared. Subsequently, the processing phase applies the context to actually perform the cryptography function. The finalization phase is responsible for cleaning up the execution context and removes security relevant data from memory. Thereby, the invocation of the cryptographic implementation differs according to the applied algorithm (AES, HMAC or CMAC).

While the AES phases are directly invoked from the application code, the operations of HMAC and CMAC are encapsulated in wrapper functions. This encapsulation in Mbed TLS is a design choice aimed at promoting code reusability, simplifying user interactions, and allowing for flexibility in the choice of cryptographic algorithms. These wrapper functions help in managing the initialization, processing, and finalization phases of cryptographic operations, ensuring the correct and

consistent execution of cryptographic functions, irrespective of the underlying algorithms being used. The wrappers also manage memory allocation for execution contexts and encapsulate the execution of functions depending on the concrete type specified as a wrapper parameter. Thereby, they ensure the correct preparation and clean up of the applied memory regions to tackle the security concerns that no information should reside in memory after cryptographic processing. This encapsulation facilitates better usability and convenience while aiding in deprecation handling, as changes in underlying algorithms or implementations can be managed within the wrapper functions. Thus, it minimizes the impact on the application code.

B. Initial Issues

We apply the process presented in Section V to conduct the analysis for the AES and HMAC implementations. The wrappers implemented in Mbed TLS use dynamic memory allocation for storing context data. Thereby, the allocation and de-allocation is performed via standard memory allocation operations `malloc`, `calloc` and `free`. However, dynamic memory management is known to be a challenging task for static timing analyses [8]. Moreover, coding guidelines focusing on safety-critical systems strongly discourage the use of dynamic memory allocation [27], [28]. However, from the security perspective the modification of the library implementation is strongly discouraged without a detailed examination of the security impacts. Hence, **we see no reasonable option to avoid the usage of heap memory for the application of the wrapper functionality in a realistic scenario.**

Thus, the wrapper implementations of the Mbed TLS library in the current shape are not worst-case timing analysable by the applied state-of-the-art static analysis tool. Hence, in favour of a successful timing analysis of the complete cryptographic primitive including the wrapper invocation, we decided to adapt the allocation mechanism of the Mbed TLS HMAC implementation. Thus, for the rest of the paper we apply a workaround memory management implementation which allocates memory statically. We denote this variant *HMAC (adapted)*. Though, wrapper can be security relevant, this adaptation is not verified according to security implications since a detailed security analysis including the evaluation of all potential flaws is out of scope of this study.

Figure 3 presents the number of issues reported from aiT after a first run of the analysis without any additional information provided. The tool basically reports the three issue categories *unresolved execution paths*, *unbounded loops* and *unknown memory accesses*. These categories correspond to the problems solving steps presented in Section V-C. However, it is not possible to automatically decide the number of undetected infeasible paths in the code under analysis. Hence, solving problems with infeasible code cannot be facilitated with automatically generated information.

CMAC is the only investigated implementation that exhibits unresolved execution paths. They originate from function pointers of the wrapper implementation. The function
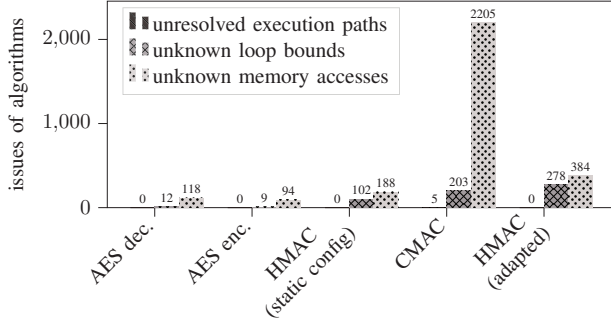
Fig. 3: Issues of the analysed algorithms stated by aiT after initial analysis run

Fig. 4: Precision of memory accesses after initial analysis run

pointers are used to call specific functions according to the concrete type and mode selected for the cryptography by passing the corresponding parameter value. Though it is possible to statically configure the library such that unused functions are not included, the function pointers remain in code and their target must be assigned to aiT with annotations.

All investigated implementations exhibit loops that cannot be statically upper bounded without further information. Thereby, the AES implementations show the lowest number of unbounded loops whereas HMAC (adapted) includes most of them. Thereby, in HMAC the hash generation algorithms like SHA-256 show a significant number of such loops. The static configuration of HMAC excludes 4 out of 5 hash methods from the binary. Hence, the number of unbounded loops of the statically configured HMAC is significantly lower than the regular one. Next to unbounded loops a similar issue is the presence of recursion. We recognized indirect recursion within the cipher update routine of CMAC. Providing a safe execution time upper bound would mean to discover and annotate the maximum number of recursive calls similar to loop bounds.

All investigated implementations exhibit memory accesses that cannot be determined statically to target a single address or a range of addresses. The HMAC implementations provide higher numbers of unknown accesses than AES. Thereby, both HMAC variants show a significant difference because of the exclusion of various code segments. In terms of stated unknown memory accesses CMAC exhibits the highest number of all investigated implementations. Moreover, the revealed number is a multiple of the numbers of the other implementations. Amongst others this is caused by the presence of different types of executions and the inability to statically decide the actually applied type. Each of those types reveals a vast amount of unknown memory accesses.

Figure 4 shows a statistic of the memory accesses of each analysed implementation. Thereby, the accesses differ according to the precision with which the analysis is able to determine the target addresses. When regarding the cumulative number of non-exact accesses (*nearly exact*, *imprecise* and *unknown*) the AES implementations exhibit small numbers
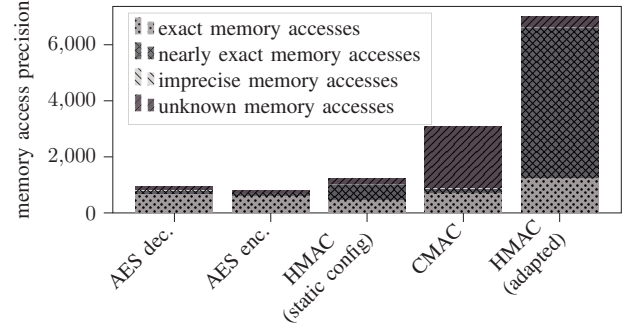
Listing 1: examples for the usage of annotation statements

```
routine "func1" {
    enter with: reg("r2") = 3;
}
routine "func2" instruction -> 128 bytes {
    accesses: 0x10000000 to 0x1000ffff;
}
instruction "func3" -> call("func4", 1) {
    enter with: user("len") = 56;
}
loop "func4.L1" {
    bound: 0 .. user("len");
}
area 0x8005cfc to 0x8005cff contains data;
```

compared to the authentication implementations. The CMAC lies in between both HMAC variants. The differentiation according to the precision reveals another distinction between symmetric cryptography and authentication. The percentage of exactly located memory accesses compared to the total number of accesses of one implementation is significantly higher for AES than for HMAC and CMAC. Hence, there are more imprecise or unknown accesses for the authentication implementations. Thereby, CMAC and HMAC clearly show different types of imprecision. While CMAC mostly shows unknown memory accesses, HMAC mostly consists of almost exact accesses.

C. Annotations

1) Annotations types: We distinguish four types of annotations which are mainly used in the analysis process. They distinguish according to the type of code they refer to. Listing 1 illustrates the different types of annotations and their usage in our analysis on a basis of examples.

One annotation type refers to the call of a routine. The first annotation in Listing 1 declares that func1 always enters the execution with value 3 in processor register r2. The second routine annotation example specifies that the instruction located 128 bytes after the beginning of routine

`func2` executes a memory access that target an address within the range of `0x10000000` to `0x1000ffff` which in our case refers to program SRAM.

Another type of annotation refers to an actual instruction. Routine and instruction annotation types can both refer to an instruction within code. Hence, they can be used for the same intention. Typically, in our analysis routine annotations ensure that a processor register or memory location contains a certain value or that accesses target a certain address range. In contrast, an instruction statement normally defines a value that is used in deeper levels of the call hierarchy to distinguish different calling contexts or to directly state a value such as message length that depends on the calling context. The instruction annotation of Listing 1 refers to the first call of `func4` within `func3` and assigns a variable *len* with 56 at the entrance of `func4`. The variable only lives in the analysis environment and is used in annotations deeper in the call hierarchy.

The third type of annotation directly points to a loop. Typically, loop annotations are used to provide loop bounds. According to the naming conventions of aiT the loop annotation of Listing 1 refers to the first loop within `func4` and reuses a previously defined variable *len*. It provides the information that the number of iterations executed for each call of that loop is within the interval 0 and the value of the variable *len*.

The forth annotation type refers to a range in the address space and enables the characterization of that area. The area example annotation states that within the address range of `0x8005cfc` to `0x8005cff` the memory only contain data and no code.

2) Provided annotations: We faced unresolved execution paths only in the CMAC implementation. However, as already mentioned, a detailed analysis of CMAC beyond the initial analysis run is out of scope of this study.

The next step is to decide if there are any uncovered infeasible paths. As mentioned above, this task requires manual code review. There are some relatively obvious infeasible paths in HMAC (adapted). These are the paths including the unused hash methods. Additionally, **there are numerous less obvious to find infeasible paths in both HMAC variants**. They accumulate to an amount of about ten paths in each variant and would exceed the WCET upper bound many times over.

We state further annotations to upper bound the number of executed loop iterations. The determination of these annotations summarizes to a significant amount of effort, since the discovered loops often iterate over elements of data structures such as arrays. More detailed, the cryptography workload often iterates over parameters such as the input text or key and initialization and finalization code often visits each memory location of a data structure (e.g., via calls to `memset`). These workload characteristics cause loop bounds in terms of the size of a data structure in bytes. Next to the initialization and finalization code the hash functions of the HMAC implementations show a lot of bounds to specify.
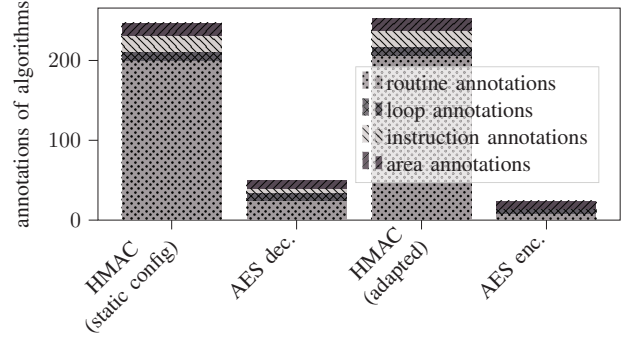

Fig. 5: Annotations stated for the implementations after the analysis is finished

Typical functions applied in initialization and finalization code are memory operations such as `memset` and `memcpy`. They are responsible for preparing memory before cryptographic processing and cleaning up memory afterwards. Since data that remain in memory after finishing cryptography are considered a security issue, these functions are mandatory for all types of cryptography implementations. Moreover, they are on the critical path for timing analysis since they have to be performed directly before and after the cryptography. Typically, loop bounds have to be set only once. Nevertheless, there is large amount of annotations necessary when functions are called from multiple calling contexts each leading to different loop bounds. Therefore, additional annotations are necessary for each context that calls the loop.

For the investigation of unknown memory accesses we only conducted a basic refinement of the analysis result by stating the address range of the targeted memory module for each unknown access. In doing so, we ensure the consideration of the correct memory access latency but accept that the analysis of the instruction cache connected to the flash memory module is destroyed for each imprecise access to the flash. However, according to the large amount of unknown accesses even this simplified process means considerable effort.

Figure 5 depicts the number of annotations made during analysis. The numbers of annotations of the AES implementations are significantly lower than the number of the HMAC implementations. Thereby, the major part of the annotations are routine annotations for deciding the target address range of unknown memory accesses. In HMAC as well as in AES the instruction annotations are mainly located in the initialization or finalization since they are mainly used to specify the number of iterations of a `memset` routine depending on the various calling contexts. Most loop annotations, in contrast, target the cryptography processing itself. Regarding HMAC, these are the loops of the SHA-256 hash processing and in AES they are needed for bounding loops of the set key routine.

Figure 6 states the precision of the analysed memory accesses after finishing the analysis. Thereby, the total number of memory accesses strongly decreased compared to the first
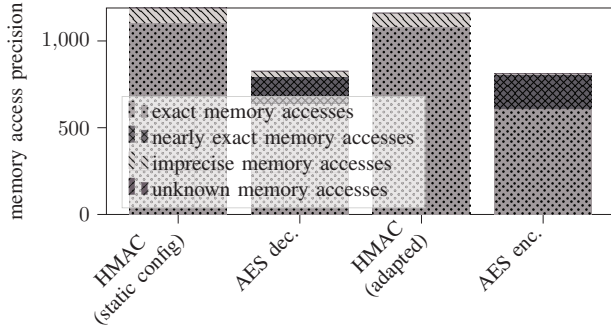
Fig. 6: Memory access precision after finishing the analysis

TABLE II: Analysed WCET upper bound and the applied annotation effort. WCET is depicted as a multiple of the measured execution time.

Algorithm	Number of Annotations	Discrepancy
AES enc.	24	1.14
AES dec.	50	1.11
HMAC (adapted)	253	2.10
HMAC (static config)	247	2.07

analysis run (see Figure 4) according to the exclusion of infeasible paths. Since we provide information to the unknown accesses which memory module is targeted, these accesses mainly turned into imprecise accesses. However, in AES several accesses could be directed to a precision of *nearly exact*.

A detailed analysis of binaries compiled with higher level of optimization is out of scope of this paper. However, we conducted some experiments to get insights on the problems when analysing optimized code. Though, an analysis of a binary compiled with -O1 only costs minor additional effort, generally the overall effort for generating annotations increases with a growing optimization level for the compile process (e.g., -O2). The number of, e.g., loop boundaries remains almost the same, but it becomes increasingly difficult to find a correct and tight upper bound. The difficulty arises from the fact that with high optimization levels it gets complicate to relate the source code to the disassembly which is finally used for the analysis. Similar difficulties arise at the investigation of target addresses for memory accesses. The drift away of assembly code from the initial source code complicates the decision of correct target addresses.

D. Gained WCET upper bounds

Table II shows the discrepancy of the analysis results from the measured execution time and the corresponding number of annotations made. The measurements were conducted through extensive testing and the maximum value being taken. The same system platform and binary was used for the static analysis and measurements, including the usage of the software and the same input sizes. In each experiment, we ran the program non-preemptively and without any operating system, i.e., bare

metal. We used the built-in hardware cycle counter to extract the execution times.

We validated the timing behaviour for different inputs of the same input size. While timing behaviour varies depending on the processor state, the execution times of the conducted experiments are not influenced by different inputs as long as the input sizes and initial processor state remain unchanged. This was expected, as the analysed algorithms from Mbed TLS are designed to be resistant to side-channel attacks that exploit input-dependent timing behaviour. In contrast, timing variations were especially noticeable when comparing the execution time with cold and warm caches.

Though, the real WCET remains unknown, the comparison of the calculated upper bound with the measured time gives a rough estimation of the overestimation gained by the analysis. The depicted table reveals that AES needs less effort to be analysed than HMAC while exhibiting more precise results (i.e., lower overestimation) at the same time. Furthermore, the overestimation of AES is bound to 14% whereas HMAC shows an execution time of more than two times of the measured execution time. While the overestimation of AES lives in the expected range, the one of HMAC clearly renders the results impractical.

In addition to the poor results, it is far more complicated for the analysis of HMAC to achieve the presented tightness. On the one hand, this is indicated by a comparison the numbers of provided annotations. On the other hand, with HMAC some annotations are way more complicated to determine than with AES. It was necessary to conduct a detailed code review to identify infeasible paths that in total caused a refinement of the discrepancy from a factor of about 7 to the presented discrepancy. Note that this review is necessary for both examined HMAC variants. We stopped our refinement at that point since the enormous effort to reach a reasonable level of tightness became sufficiently clear.

We conducted further analysis runs to investigate the reason of the remaining overestimation of HMAC. In one experiment, we unrealistically assumed that the cache accesses always hit to evaluate if the destruction of the cache analysis caused by imprecise memory accesses lead to this overestimation. We found that assuming only cache hits lead to a discrepancy factor of 1.93 for HMAC (adapted) and to a factor of 1.89 for the statically configured variant. As the HMAC discrepancy factors are 2.10 and 2.07, the analysed worst case caching effects cause at maximum about 18% of the overestimation. Other experiments that e.g., assume only aligned memory access have only minor impact on the analysis result.

Comparing both HMAC variants, the *(static config)* variant reveals a WCET upper bound that is closer to the measured execution time. In addition, this result is achieved with fewer annotations. Moreover, the saved annotations belong to the ones that cause an increased effort due to the determination of infeasible paths. Though the savings in this particular case seem marginal, they may be relevant in implementations that are closer to a feasible upper bound.

Altogether, the AES implementations of Mbed TLS are

analysable with reasonable effort. In contrast, the HMAC implementation does not show practically relevant results, even when investing an overly high amount of effort.

VII. DISCUSSION

The AES encryption and decryption implementations of Mbed TLS exhibit reasonable WCETs when analysed with aiT. Furthermore, the static configuration of the primitives facilitates the safety considerations as it excludes various infeasible paths and unbounded loops. Hence, static configurations not only improves the resource consumption but also enhances the timing analysability. We see the option to statically configure the library according to the needs of the current application as a crucial benefit that facilitates a static real-time analysis. Hence, we recommend using this option extensively.

A. Safety concerns

The investigation of the authentication primitives revealed significant difficulties for the application in hard real-time systems. These difficulties are strongly related to the wrappers applied by Mbed TLS. Typically, the wrappers use dynamic memory allocation and partly apply function pointers. Both design choices violate the recommendations of safety coding guidelines. Since the library was developed to be highly generic, manual effort is needed to determine the correct pointer destination. Depending on the context, various functions can be called by the same command. This can only be examined at runtime. Unfortunately this issue cannot be solved by statically configuring the library. Therefore, the **wrappers are not timing analysable with the analysis tool at hand in their current shape**.

The investigated security primitives exhibit a vast amount of unbounded loops often with upper bounds that depend on the calling context or the input size. Such loops mainly occur within the hash calculation of authentication primitives. However, they are also present in other parts of the primitives. The presented safety guidelines recommend a simple loop structure with an explicit integer counter that is not adapted within the loop body. This recommendation is violated for some loops e.g., when decreasing the iteration counter by a value gathered from a return value of a function called within the loop body. However, most of the loops can be refactored to fit the safety field's preferred structure. The actual adaptation of the loops as well as a detailed security analysis is future work.

The usage of recursion is related to the issue about unbounded loops. Direct as well as indirect recursion violates the recommendations of MISRA-C and "The Power of 10". In the investigated implementations, CMAC exhibits indirect recursion. Though forbidden in presented safety guidelines, aiT provides the annotation of a recursion upper bound to state the maximum number of recursive calls. However, such bounds can typically not be stated automatically. As with loops, it is necessary to determine an upper bound.

Especially the authentication mechanisms suffer from the large amount of provided annotations necessary to achieve

reasonable timing bounds. Next to the effort it also weakens the soundness of the analysis because the responsibility of the correctness of the manually provided information remains to the developer conducting the analysis. Hence, the additional information are not covered by the verified correctness of the analysis tool.

B. Security concerns

The usage of wrappers is strongly recommended in terms of security to avoid missing security relevant initialization and clean up tasks and to ensure the correct application of the library. On the other hand, an in-house adaptation of the library is strongly discouraged, since the adapted code is not part of the security evaluation process that guarantees the secure behaviour of each primitive. Hence, **each adaptation may introduce errors or vulnerabilities and thus, conflict with the recommendations of security standards**.

According to the lack of proper timing analysability of Mbed TLS primitives, one may be tempted to ignore the security industry maxim "don't roll your own crypto". This may lead to a code base certifiable according to safety standards, but it definitely violates security considerations. To illustrate this fact we provide insight to security primitives found in the real-time related benchmark suite TACLe bench, which we mentioned in Section II. This suite comprises implementations of hash algorithms MD5 and SHA1 and symmetric cryptography with AES.

An investigation in terms of security revealed that the AES benchmark and all hash implementations in TACLe are not intended for the usage in real world security scenarios. The origin of those implementations partly remains unclear, and the code base shows some obscure implementation details. Furthermore, the provided hash algorithms are outdated, since it has been shown that they are broken [46]. For that reason the code base of TACLe security implementations must be considered insecure. Hence, we see the maxim confirmed that "rolling one's own cryptography" implies an increased risk of developing insecure code either via introducing security flaws or applying outdated primitives.

C. Recommendations

Can we ensure security and safety via real-time without compromising either? We have found non-analysable cryptographic primitives in Mbed TLS and code that violates common safety guidelines. Hence, in a general case it is not possible to utilize this library in hard real-time systems. However, there are also analysable primitives. Thus, in a more specific scenario the applicability depends on the primitives that are required to satisfy the security concerns arising in a particular system under development.

The main reasons for compromising safety concerns are located in the wrapper implementations of Mbed TLS. Their usage of dynamic memory allocation and function pointers significantly complicates static real-time analysis. However, from a security point of view, they are mandatory to simplify

the application of the primitives and prevent the introduction of security flaws through careless usage of the library.

If these issues can be solved securely, we see that security primitives of Mbed TLS are able to fulfil the implementation requirements for passing the safety certification processes. Thereby, we recommend putting special focus on the wrapper implementations. We propose to provide less generic wrappers to eliminate hard-to-analyse function pointers. However, we see that this approach would come at the price of an enlarged set of wrappers. Furthermore, the usage of static memory instead of dynamic memory allocation would be another important advantage in terms of safety compliance. But those adaptions must be subject to a detailed security evaluation to prevent the introduction of subtle security flaws. Apart from the adaptation of wrappers, we see potential for improvement of the real-time capability in providing loop counters and upper bounds that are compliant to the safety guidelines.

VIII. CONCLUSION

This study investigated the conflict of interests between safety and security concerns in embedded hard real-time systems. Our question was whether we can serve the requirements of both fields without compromising either. We examined this question by deriving the WCET for state-of-the-art symmetric cryptography and authentication provided by one of the de-facto standard libraries for embedded systems, Mbed TLS. Furthermore, we investigated its code base with regard to compliance on safety-related coding guidelines. Finally, we assessed the relevant aspects when security concerns must be considered in the safety-related context.

According to the obtained results safety and security are not combinable without compromising either in the general case. Our investigations revealed that large parts of the code base of Mbed TLS are not compliant with common safety coding guidelines. Furthermore, important parts of the library are not timing analysable with the state-of-the-art tool aiT for static WCET analysis. However, few primitives of Mbed TLS are applicable to hard real-time systems. Hence, in rare special cases their application may be a suitable option.

Altogether, simultaneously complying with safety and security concerns remains an open problem. Hence, for future work we propose to build a library for cryptographic primitives that suits the requirements of both fields. The adaptation of an existing commonly used library like Mbed TLS is also a viable option. Either way the proposed library should be compliant to common safety and security standards and provide a vast set of cryptographic primitives to provide suitable cryptography for a wide range of embedded applications.

REFERENCES

[1] IEC, *IEC 61508:2010 CMV*. VDE VERLAG GmbH, 2010. [Online]. Available: https://www.vde-verlag.de/iec-normen/217360/iec-61508-2010-cmv.html

[2] RTCA, *DO-178B*. VDE VERLAG GmbH, 1992. [Online]. Available: https://www.iso.org/standard/68383.html

[3] D. Kästner and C. Ferdinand, "Safety Standards and WCET Analysis Tools," in *Embedded Real Time Software and Systems (ERTS2012)*, Toulouse, France, Feb. 2012. [Online]. Available: https://hal.science/hal-02192406

[4] DIN, *IEC 62443-4-2 VDE 0802-4-2:2019-12*. VDE VERLAG GmbH, 2019. [Online]. Available: https://www.vde-verlag.de/normen/0800631/din-en-iec-62443-4-2-vde-0802-4-2-2019-12.html

[5] R. Zabicki, *Practical Security – Simple Practices for Defending Your Systems*. Pragmatic Programmers, LLC, The, 2019.

[6] TrustedFirmware, "Mbed TLS," 2023, Accessed: 2023-05-05. [Online]. Available: https://www.trustedfirmware.org/projects/mbed-tls/

[7] *XMC4500 Microcontroller Series for Industrial Applications - Reference Manual*, Infineon Technologies AG, 2016, edition 2016-07. [Online]. Available: https://www.infineon.com/dgdl/Infineon-xmc4500_rm_v1.6_2016-UM-v01_06-EN.pdf?fileId=db3a30433580b3710135a5f8b7bc6d13

[8] G. Gebhard, C. Cullmann, and R. Heckmann, "Software Structure and WCET Predictability," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, ser. OpenAccess Series in Informatics (OASIcs), P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, Eds., vol. 18. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, pp. 1–10. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2011/3083

[9] AbsInt Angewandte Informatik GmbH, "AbsInt aiT," 2023, Accessed: 2023-05-05. [Online]. Available: https://www.absint.com/ait/index.htm

[10] ISO, *ISO 26262-1:2018*. VDE VERLAG GmbH, 2018. [Online]. Available: https://www.iso.org/standard/68383.html

[11] S. Mohan, "Worst-Case Execution Time Analysis of Security Policies for Deeply Embedded Real-Time Systems," *ACM SIGBED Review*, vol. 5, no. 1, Jan. 2008. [Online]. Available: https://doi.org/10.1145/1366283.1366291

[12] F. Mueller, "Challenges for cyber-physical systems: Security, timing analysis and soft error protection," in *High-Confidence Software Platforms for Cyber-Physical Systems (HCSP-CPS) Workshop, Alexandria, Virginia*, vol. 6, 2006. [Online]. Available: https://ptolemy.berkeley.edu/projects/chess/pubs/601/cps-security-challenges.pdf

[13] Y. Ke, X. Xia, and S. Babaie, "Timed Automaton-Based Quantitative Feasibility Analysis of Symmetric Cipher in Embedded RTOS: A Case Study of AES," *Security and Communication Networks*, vol. 2022, p. 4118994, 2022. [Online]. Available: https://doi.org/10.1155/2022/4118994

[14] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research," in *16th International Workshop on Worst-Case Execution Time Analysis*, Toulouse, France, 2016. [Online]. Available: https://hal.science/hal-02610690

[15] I. Yarza, I. Agirre, I. Mugarza, and J. Perez Cerrolaza, "Safety and security collaborative analysis framework for high-performance embedded computing devices," *Microprocessors and Microsystems*, vol. 93, p. 104572, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933122001247

[16] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache Side-Channel Attacks and Time-Predictability in High-Performance Critical Real-Time Systems," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3195970.3196003

[17] A. Carelli, A. Vallero, and S. Di Carlo, "Performance Monitor Counters: Interplay Between Safety and Security in Complex Cyber-Physical Systems," *IEEE Transactions on Device and Materials Reliability*, vol. 19, no. 1, pp. 73–83, 2019.

[18] C. Zimmer, B. Bhat, F. Mueller, and S. Mohan, "Time-Based Intrusion Detection in Cyber-Physical Systems," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 109–118. [Online]. Available: https://doi.org/10.1145/1795194.1795210

[19] B. McDonald and F. Mueller, "T-SYS: Timed-Based System Security for Real-Time Kernels," in *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*, May 2022, pp. 247–258.

[20] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing Analysis of a Protected Operating System Kernel," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 339–348.

[21] A. Horga, "Performance and Security Analysis for GPU-Based Applications," Ph.D. dissertation, Linköping University, Software and

[22] M. Völp, D. Kozhaya, and P. Verissimo, "Facing the Safety-Security Gap in RTES: the Challenge of Timeliness," 2017. [Online]. Available: https://orbilu.uni.lu/bitstream/10993/34057/1/Timeliness-SafeSecGap.pdf

[23] T. Wollinger, J. Guajardo, and C. Paar, "Cryptography in embedded systems: An overview," *Proc. Embedded World*, pp. 735–744, 2003.

[24] C. Manifavas, G. Hatzivasilis, K. Fysarakis, and K. Rantos, "Lightweight cryptography for embedded systems–a comparative analysis," in *International Workshop on Data Privacy Management*. Springer, 2013, pp. 333–349.

[25] T. Silde, "Comparative study of ECC libraries for embedded devices," *Norwegian University of Science and Technology, Tech. Rep*, 2019.

[26] S. Kerckhof, F. Durvaux, C. Hocquet, D. Bol, and F.-X. Standaert, "Towards green cryptography: a comparison of lightweight ciphers from the energy viewpoint," in *Cryptographic Hardware and Embedded Systems–CHES 2012: 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings 14*. Springer, 2012, pp. 390–407.

[27] The MISRA Consortium Limited, "Publications," 2023, Accessed: 2023-10-06. [Online]. Available: https://misra.org.uk/publications/

[28] G. Holzmann, "The Power of 10: Rules for Developing Safety-Critical Code," *IEEE Computer*, vol. 39, pp. 95–97, Jun. 2006.

[29] JPL Jet Propulsion Laboratory, *JPL Institutional Coding Standard for the C Programming Language*. California Institute of Technology, 2009. [Online]. Available: http://everyspec.com/NASA/NASA-JPL/JPL-D-60411_VER-1_32832/

[30] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: Association for Computing Machinery, 1977, pp. 238–252.

[31] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem – overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, May 2008.

[32] Infineon Technologies AG, *XMC4500 Microcontroller Series for Industrial Applications - Data Sheet*, Infineon Technologies AG, 2023, edition 2023-04. [Online]. Available: https://www.infineon.com/dgdl/Infineon-XMC4500-DataSheet-v01_06-EN.pdf?fileId=5546d46254e133b40154e1b56cbe0123

[33] AbsInt Angewandte Informatik GmbH, *aiT for ARM - Factsheet*, AbsInt Angewandte Informatik GmbH, 2023, release 23.04i, [Online; accessed 28-September-2023]. [Online]. Available: https://www.absint.com/factsheets/factsheet_ait_arm_web.pdf

[34] *XMC 4500 Relax Kit & XMC 4500 Relax Lite Kit - Board User's Manual*, Infineon Technologies AG, 2014, edition 2014-01-13. [Online]. Available: http://www.infineon.com/dgdl/Board_Users_Manual_XMC4500_Relax_Kit-V1_R1.2_released.pdf?fileId=db3a30433acf32c9013adf6b97b112f9

[35] Q. Meng and L.-T. Hsu, "Integrity for autonomous vehicles and towards a novel alert limit determination method," *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, vol. 235, no. 4, pp. 996–1006, Oct. 2020.

[36] Linaro Limited, "Mbed TLS," 2009, Accessed: 2023-10-25. [Online]. Available: https://github.com/Mbed-TLS

[37] The OpenSSL Project, "OpenSSL," 1998, Accessed: 2023-10-25. [Online]. Available: https://www.openssl.org/

[38] T. Ouska, "wolfSSL/wolfCrypt," 2006, Accessed: 2023-10-25. [Online]. Available: https://www.wolfssl.com/

[39] Intel, "wolfSSL/wolfCrypt," 2015, Accessed: 2023-10-25. [Online]. Available: https://github.com/intel/tinycrypt

[40] N. Mavrogiannopoulos and S. Josefsson, "GnuTLS," 2000, Accessed: 2023-10-25. [Online]. Available: https://www.gnutls.org/

[41] Team libtom, "LibTomCrypt," 2001, Accessed: 2023-10-25. [Online]. Available: https://www.libtom.net/LibTomCrypt/

[42] GitHub, Inc., "Saving repositories with stars," 2024, Accessed: 2024-02-28. [Online]. Available: https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars

[43] A. Khlebnikov, *Demystifying Cryptography with OpenSSL 3.0*, J. Adolfsen, Ed. Birmingham: Packt Publishing Limited, 2022.

[44] F. A. Stancu, C. D. Trancă, M. D. Chiroiu, and R. Rughiniş, "Evaluation of cryptographic primitives on modern microcontroller platforms," in *2018 17th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, Sep. 2018, pp. 1–6.

[45] NIST, US, "National Institute of Standards and Technology. Federal information processing standards (FIPS 180-2). Announcing the Secure Hash Standard (August 2002)," 2004, Accessed: 2023-10-09. [Online]. Available: https://csrc.nist.gov/pubs/fips/180-2/upd1/final

[46] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The First Collision for Full SHA-1," in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 570–596.