Identification and Verification of Security Relevant Functions in Embedded Systems Based on Source Code Annotations and Assertions

Johannes Loinig, Christian Steger, Reinhold Weiss, Ernst Haselsteiner

Identification and Verification of Security Relevant Functions in Embedded Systems Based on Source Code Annotations and Assertions

Johannes Loinig[1], Christian Steger[1],
Reinhold Weiss[1], and Ernst Haselsteiner[2]

[1] Institute for Technical Informatics, Graz University of Technology, Graz, Austria
Email: {loinig, steger, rweiss}@tugraz.at
[2] NXP Semiconductors Austria GmbH, Gratkorn, Austria
Email: ernst.haselsteiner@nxp.com

Abstract. Most modern embedded systems include an operating system. Not all functions in the operating systems have to fulfill the same security requirements. In this work we[3] propose a mechanism to identify and maintain functions that have to meet strict security needs. This mechanism is based on annotations representing security constrains and assertions to check these security annotations during the verification phase of the system under development.

1 Introduction

Every modern operating system (OS) is split in hundreds of functions. Not every function has to fulfill the same security requirements. For example, some of them must not leak secrets such as cryptographic keys. Others must be timing invariant for all kinds of inputs. Finally, every OS has many functions that do not need to fulfill special security requirements. Of course they have to work properly and must not open back doors to potential attackers (e.g. by buffer overflows) but actually we do not regard this as a special security constraint. In fact, wee consider this property as normal. In the remainder of this work we call them *security neutral functions*.

Implementing all functions on the highest security level (with respect to all possible security countermeasures) is not feasible and not necessary.The development cost would be too high as implementing secure functions is more time consuming. The performance of a system would be too slow as additional security usually causes a computational overhead. To compensate this faster hardware would be necessary which again increases the costs. Finally, the executable code would increase which is a significant cost factor if the code is masked into the ROM.

If a function has to meet a selected security constraint (SC), e.g. to check program flow integrity, it must be ensured that every subfunction which is called also has to meet the same SC. If not, this may raise a weak point in the chain

[3] This paper is a result of the *HiPerSec* project which is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the contract FFG 816464.

of trust representing the call hierarchy of the functions. As the security of the system can only be as strong as the weakest point, the developers must take care that they do not use security neutral functions for operations which are expected to provide security.

The aim of our work is to provide a mechanism to differentiate between functions that have to fulfill security requirements and functions that do not. To do so, we use in-line source code annotations to mark security relevant parts of the source code. Our proposed mechanism allows a design tool to verify these security annotations during the development process and thus helps developers to increase and maintain the security and performance of complex embedded systems.

2 Related Work

Security has to be considered from the beginning of the product life cycle. The implementation of security elements have to be done in a well organized way [1]. Kocher et al. state in [2] that it is a problem that the implementation is very often done by security experts who are the only people in a development team that really understand the security requirements. The reason is that a system's security is deeply rooted in the complete development process and cannot be implemented by covering some few selected points that were identified to have to be secure. A cryptographic function, for example, can be implemented in a perfect way but will be simply useless if the private keys are handled in an insecure way while loading them from memory.

Furthermore, [2] states that a formal verification of programs with realistic complexity is not feasible today. Good engineering practice which covers all software artifacts as security objectives is necessary to develop secure products. Analysis tools and techniques to map security requirements to solutions and explore trade-offs are needed.

In [3] existing static code analysis tools for security checks are summarized. There exist a lot of tools checking for vulnerable constructs, proper usage of types and values, race conditions, and so on. However, the authors state that there is still a deficit for checkers that include relationships between functions.

A model-based planning strategy for security requirements is described in [4]. A high-level model instead of source code is used to verify the formal properties of functional and security requirements.

Source code annotations are declarative information for runtime entities and are supported by several modern programming languages [5] and software frameworks [6]. They can be evaluated by tools during the development process or by the runtime environment during execution time. One example is `@deprecated` in Java which defines that a function should not be used anymore. The Java compiler generates a warning if a deprecated function is used. In this work we use annotations to define which SCs were considered during the implementation of the annotated software functions.

Our proposed mechanism can be used to analyze the final source code of the product. It verifies chains of trust through the whole software of an embedded system across boundaries of functions and software layers. To do so, we use well

known and established tools like source code annotations and assertions during the development process.

3 Identification and Verification of Security Relevant Functions

The basis of our proposed concept is that a developer of a security relevant function does not need to think about the security status of called subfunctions. The developer knows which SCs are defined for a function that has to be implemented. He or she can implement the function without risk that used subfunctions will not be able to provide the necessary SCs. Thus, the development process is widely concentrated on the new function which makes development easier, faster, and more secure.

Figure 1 shows the basic workflow of our proposal. Different developers implement functions according to their functional requirements and security requirements. They annotate the newly implemented functions with appropriate SCs. We define the meaning of SCs in more detail in Section 3.2. When the system's modules become integrated a design tool checks the security properties of all used functions, including functions in external libraries. The design tool reports security violations if security properties of functions can not be fulfilled by called subfunctions.

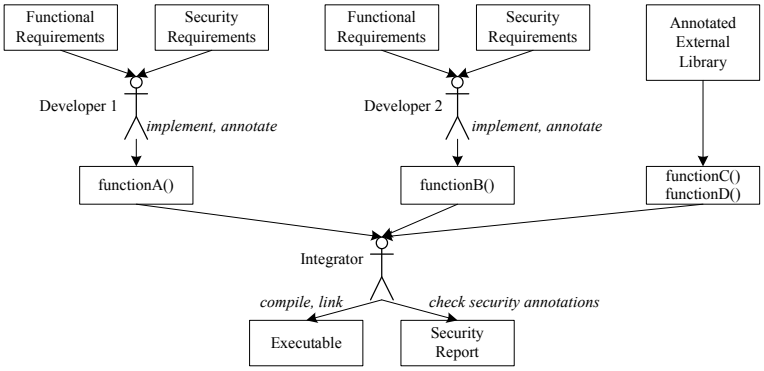


Fig. 1. The workflow of our proposed concept.

Furthermore, our proposed concept allows identification of subfunctions that may have a too high security status. Such a function implements SCs that are not required by the calling function. This may be the case intentionally, if the subfunction requires the SC, or by accident if a subfunction was reused. In latter case, the appropriate usage of a function with a lower security status instead, can increase the system's performance without security drawbacks.

3.1 Assigning Security Constraints

Every function is annotated with the SC it implements. The same SC is supposed to be provided by called subfunctions. If not, the chain of trust which is implicitly given by the function call hierarchy is broken. This is an indication for a potential security gap in the system. If the subfunction provides a higher security level this is an indicator for potential performance optimizations.

The concept is shown in Figure 2. As can be seen `functionB()` breaks the chain of trust and `functionC()` provides a higher security status which may cause unnecessary performance reduction because the additional but unnecessary security may slow down the function's execution.

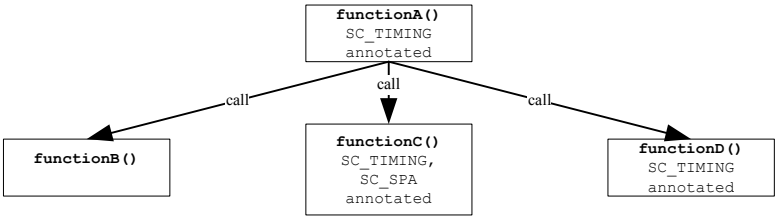


Fig. 2. Simple example of functions which are annotated with security constraints.

3.2 Definition of Security Constraints

The proposed concept is not restricted to a certain number or nature of security constraints. A threat model or attack tree can be used to deduce necessary SCs at the beginning of the system's design phase. We derived a list of SCs for the software implementation of a smart card from attacks described in [7–10]. The list below is not exhaustive but should indicate how the concept of SCs works.

Timing Attack (SC_TIMING): A function which is annotated with this annotation has to provide timing which is independent from any input data to avoid timing attacks.

Simple/Differential Power Analysis (SC_SPA, SC_DPA): changes of the system's power state can be observed externally and must be avoided.

Differential Fault Analysis (SC_DFA): fault injection can not be prevented by software but the function can provide e.g. recalculation and comparison of results to detect potential injected faults.

Perturbation Attacks (SC_PERT): an annotated function must check the integrity of data and the integrity of the program flow to detect disturbances of the normal software behavior (e.g. caused by laser beam attacks).

The detailed meaning and implementation of an SC can vary between different systems and different functions. In a real development process the SCs have

to be clearly defined and communicated to the developers. Assigning SCs should be followed by a code review done by a different developer to ensure that the system is not corrupted by miss-assigned annotations.

3.3 Annotation based Identification of Security Relevant Functions

The basis of our proposed mechanism is a function call tree. A call tree shown in Figure 3 (a) is extended to a tree representing the chains of trust of the calling functions, shown in Figure 3 (b). When the call tree is transformed to the tree of trust chains each node is replaced by a node including all SCs assigned to the corresponding function.

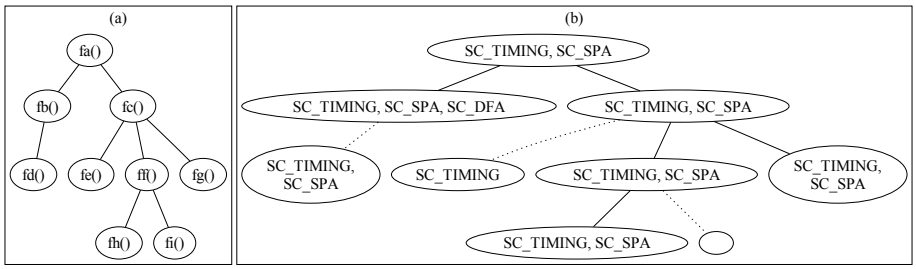


Fig. 3. A call tree in (a) and after its transformation to a tree of trust chains in (b).

A tool runs through all the nodes in the extended call tree. Every node must include all SCs of its parent. If a broken chain of trust is found the tool reports the security constraint violation.

The generation of the call tree can be done by static code analysis or during execution of a use case. It may be difficult in some situations to setup a capable code analysis if different layers of software are used. If, for example, the system is based on a virtual machine (VM), it may be difficult to maintain function calls from the software running on the VM to the underlying software layers. An example is a Java Card [11], a smart card including a Java VM. Such a system can easily be split in four different software layers: the Java application, the Java OS, the native hardware-independent software written in C, and the assembler functions. In such a case, an execution based generation of a call tree may be more promising than a static code analysis through software layers implemented in different programming languages.

3.4 Assertion based Verification of Security Relevant Functions

For system verification we propose an assertion based mechanism that checks the security constraints during the verification of the system. When a subfunction is called, all SCs of the calling function are passed to the subfunction to be checked. The subfunction provides an assertion and verifies if all necessary SCs are implemented.

The SCs are only used during the system development and verification process. They are not needed anymore when the system operates in the field. Therefore, neither annotations nor assertions have to be included in the final product.

4 Implementation

We implemented our concept on the basis of a real but simulated Java Card operating system. As application we chose the JavaPurse application included in the Java Card Development Kit [12]. So far, the OS does not provide any annotations of SCs. Thus, we annotated the security relevant high-level functions of the JavaPurse `processVerifyPIN()`, `processInitializeTransaction()`, and `processCompleteTransaction()`. All three methods are called in `process()` which is called for every command that is received by the JavaPurse application. We identified these three functions as security relevant because they check if the card holder is able to provide the right PIN and initialize respectively complete the payment transaction.

The used simulation environment is a SystemC [13] model of the smart card hardware. We executed one payment transaction and recorded the broken chains of trust which emerge from our annotations in the JavaPurse application. As only functions of the Java Card application were annotated, the mechanism reports all functions in the OS that are called during the payment transaction.

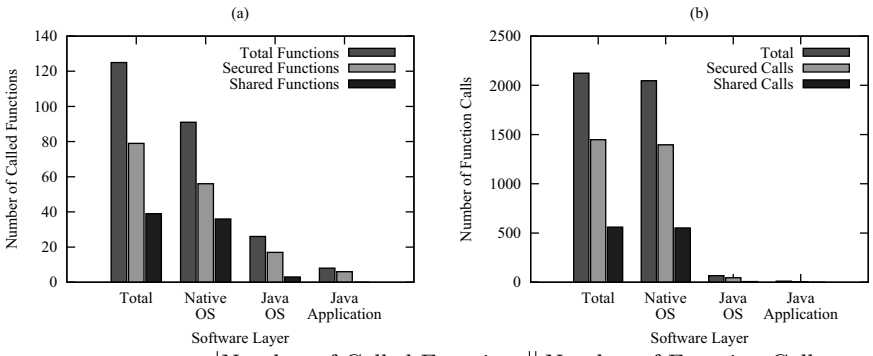
The SystemC model uses an annotation stack which is filled with annotations that have to be implemented by the current called function. All entries of the stack are checked for every simulated function call. Annotations of functions are passed via a Java API to a special function register in the SystemC model of the smart card. The evaluation of the annotations is done in the simulated call-instructions and return-instructions.

5 Experimental Results

Figure 4 (a) shows the number of functions which were called during one payment transaction of the JavaPurse application. 125 functions were called in total, 91 of them are in the native OS layer, 26 in the Java OS layer, and 8 in the application layer. 63% of them were noted as included in a broken chain of trust, which means that these 79 functions should be checked if they fulfill the security requirement. Notice that for our proof of concept evaluation there was no need to define this requirement in detail. 49% of the functions that have to fulfill special security requirements are also used in a context where the security requirement is not needed (named as shared functions in the diagram). This opens a significant potential for performance optimizations.

Additionally, Figure 4 (a) shows the partitioning of the used functions in the software layers. As expected, most security relevant functions are implemented in the native OS layer. We think that identification of these functions is especially important as they are naturally not secured by the Java VM.

Figure 4 (b) shows the function calls which were executed during the payment transaction. 2124 function calls were executed in total, 2046 in the native OS, and respectively only 67 and 11 function calls in the Java OS and application layer.



	Number of Called Functions			Number of Function Calls		
	Total	Secured	Shared	Total	Secured	Shared
Total	125	79	39	2124	1447	559
Native OS	91	56	36	2046	1395	552
Java OS	26	17	3	67	46	7
Java Application	8	6	0	11	6	0

Fig. 4. Results Diagrams and Table. (a) The number of function calls, (b) the number of called functions.

68% of all function calls were marked as security relevant by our mechanism. 39% of them were also done in a security neutral state of the system. As can be seen in the diagram the number of Java function calls is minimal in comparison to the native function calls. Therefore, we can argue that the overhead from our additional Java API, passing the annotations through the VM to the SystemC model, is irrelevant.

Our modification of the SystemC model, checking the functions' annotations, did not cause any significant performance reduction. This is shown in Figure 5. A simple demo program executed 1000, 2000, and 3000 function calls with 4 annotations, 8 annotations, and disabled annotation mechanism. We normalized the results to 100% for 1000 functions without annotations. According to the results in Figure 5 we can argue that the simulation based verification performance does not suffer drastically from our proposed assertion mechanism.

6 Conclusion

In this work we proposed an annotation based method to identify security relevant functions in complex software of embedded systems. This ensures that security constraints of functions are not violated by their called subfunctions and hence increases the system's security. In addition we presented how the same mechanism identifies functions which should be considered for performance optimizations as they do not have to fulfill special security constraints all the time. 49% of all called functions in our demo application were potential targets for such optimizations.

We verified our concept by a proof-of-concept implementation checking assertions for security constraints during simulation time of a SystemC model. Our

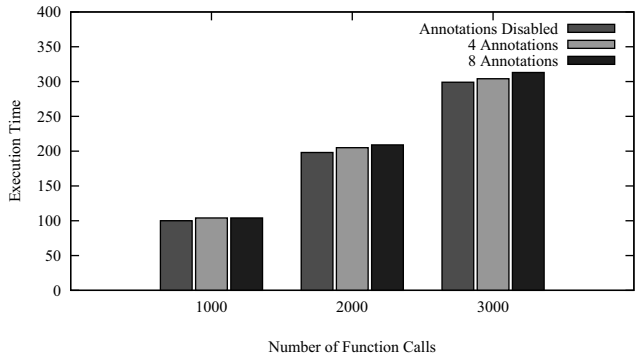


Fig. 5. Simulation Performance of the Annotation/Assertion Mechanism.

tests showed that this can be done without significant performance overhead during the simulation of complex systems.

Summarizing this, our approach can be used to optimize the trade-off between security and performance of a complex embedded system.

References

1. Schaumont, P., Verbauwhede, I.: Domain-specific codesign for embedded security. Computer 36(4) (April 2003) 68–74
2. Kocher, P., Lee, R., McGraw, G., Raghunathan, A.: Security as a new dimension in embedded system design. (2004) 753–760
3. Chess, B., McGraw, G.: Static analysis for security. Security & Privacy, IEEE 2(6) (Nov.-Dec. 2004) 76–79
4. Bryl, P.V., Bryl, V., Massacci, F., Mylopoulos, J., Zannone, N.: Designing Security Requirements Models through Planning. In: In Proceedings of CAiSE'06, 2006, Springer (2006) 33–47
5. Ernst, M.D.: Type Annotations Specification (JSR 308) (November 2008)
6. Newkirk, J., Vorontsov, A.: How .NET's custom attributes affect design. Software, IEEE 19(5) (Sep/Oct 2002) 18–20
7. Eagles, K., Markantonakis, K., Mayes, K.: A comparative analysis of common threats, vulnerabilities, attacks and countermeasures within smart card and wireless sensor network node technologies. Lecture Notes in Computer Science 4462 (2007) 161
8. Sere, A.A., Iguchi-Cartigny, J., Lanet, J.L.: Automatic detection of fault attack and countermeasures. In: WESS '09: Proceedings of the 4th Workshop on Embedded Systems Security, New York, NY, USA, ACM (2009) 1–7
9. Rankl, W., Effing, W.: Smart Card Handbook. John Wiley & Sons, Inc., New York, NY, USA (2003)
10. Common Criteria: Application of Attack Potential to Smartcards Version 2.7 Revision 1 (March 2009)
11. Sun Microsystems, Inc.: Java Card Platform Specification 2.2.2
12. Sun Microsystems: Java Card Development Kit. `http://java.sun.com/javacard/devkit/` (March 2006)
13. IEEE: Open SystemC Language Reference Manual. (December 2005)