COGITO: Code Polymorphism to Secure Devices

Damien Couroussé[1], Bruno Robisson[2], Jean-Louis Lanet[3], Thierno Barry[1], Hassan Noura[1], Philippe Jaillon[4] and Philippe Lalevée[4]

[1]*Univ. Grenoble Alpes, F-38000 Grenoble, France CEA, LIST, MINATEC Campus, F-38054 Grenoble, France*

[2]*Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA) Gardanne, France*

[3]*University of Limoges, 123 rue Albert Thomas, 87000 Limoges, France*

[4]*École Nationale Suprieure des Mines de Saint-Etienne (ENSM.SE) Saint-Etienne, France*

{1, 2, 4, 5}th author@cea.fr, 3rd author@xlim.fr, {6, 7}th author@ensmse.fr

Keywords: Code Polymorphism, Runtime Code Generation, Physical Attacks, Embedded Devices

Abstract: In this paper, we advocate the use of code polymorphism as an efficient means to improve security at several levels in electronic devices. We analyse the threats that polymorphism could help thwart, and present the solution that we plan to demonstrate in the scope of a collaborative research project called COGITO. We expect our solution to be effective to improve security, to comply with the computing and memory constraints of embedded devices, and to be easily generalisable to a large set of embedded computing platforms.

1 Introduction

Electronic devices have grown a huge interest for the average user thanks to the tremendous computation power and data storage that it is possible to put in a few square centimetres of silicon. Everybody has now the capability to put a lot of its personal and professional information in such devices, and the confidentiality of the 'secret' kept in electronic devices is of huge importance. As a consequence, security of such devices is one of the main concerns, on an equal footing with user interfaces and connectivity. Therefore, when thinking about security issues in electronic devices, *smart card* or *Java Card* are the words that first come to mind for historical reasons. But today, the landscape of electronic devices is composed of computing platforms of very different kinds. Hence, security is now an issue for a very large set of embedded devices, from nodes in wireless sensor networks to smartphones.

All of these electronic devices can be subject to "physical attacks" which are generally split in two steps, thereafter called 'first step' attacks and 'second step' attacks. 'First step' attacks consist in getting general information about the functioning of the devices: identifying which security functions are implemented, when they are launched and on which hardware blocks they are executed. The term 'reverse engineering', in the large meaning of the word, is also commonly used to refer to this step of the attack. In this step, the attacker can use three different kinds of techniques. The first one consists in getting information about the chip design by direct inspection of its hardware structure. This inspection may be performed by using any kind of imaging techniques or by using destructive means such as abrasion, chemical etching or focused ion beam. The second kind, called *side channel attacks*, consists in observing some physical characteristics (such as power consumption, electromagnetic radiation, response time, etc.) which are modified while the circuit is active. The third technique, called *fault attacks*, consists in disrupting the circuit's behaviour by using laser beams, voltage or clock glitches, electromagnetic pulses, etc. In the 'second step', the attack is focused on one or several security functions in order to bypass them (Barbu et al., 2010), to recover the details of their implementation (Novak, 2003) or to recover the manipulated data (Genkin et al., 2013). These 'second step' attacks are very powerful but always rely on a very precise spatial and temporal control of the target. In particular, the success of a fault (resp. side channel) attack depends on the time precision required to inject the fault (resp. measure the physical characteristics) at the right time and on the right area during program execution.

Many protections (also called 'ad hoc') have been proposed to counter all these 'second step' attack schemes. But surprisingly, few protections have been proposed against 'first step' attacks, even if these ones constitute a crucial stepping stone for the success of an attack. We assume that polymorphism, which consists in regularly changing the behaviour of a secured component at runtime while maintaining unchanged its functional properties, can make attackers' lives more difficult. Our hypothesis is that polymorphism can tackle security issues at several levels: it increases the difficulty of reverse engineering, it provides dynamically changing temporal and spatial properties that also increase the difficulty of 'second step' physical attacks. Moreover, we assume that the use of polymorphic code is a realistic objective even for resource-constrained embedded devices, and illustrate how we plan to achieve this thanks to a tool for runtime code generation that fulfils the memory and computing constraints of embedded systems.

To demonstrate our hypothesis, the rest of this paper is organised as follows. Section 2 presents an overview of the general principles of protections and of the uses of polymorphism in the context of security. Section 3 introduces the solution envisioned in this paper and provides a simple illustrative case. Section 4 concludes.

2 State of the art

We make the distinction between general protections, which make reverse engineering of the target more difficult to achieve in 'first step' attacks, and ad hoc protections, which are specific defensive mechanisms implemented against a known attack usually performed as a 'second step' attack.

2.1 Ad hoc protections

Many protections have been proposed to counter 'second step' hardware attacks. Some protections (called 'sensors') give information about the state of the system either by measuring light (Dutertre et al., 2013), voltage (Zussa et al., 2014), frequency or temperature or by detecting errors during computations. This detection is generally based on spatial redundancy, temporal redundancy or information redundancy. Several mechanisms are also proposed to detect a modification of the execution flow of a program (Petroni and Hicks, 2007), or to avoid the effects of instruction skips caused by fault attacks (Moro et al., 2014). In some cases, an additional hardware block is dedicated to this task (Arora et al., 2005). To reduce sensitivity to side channel attacks, 'noise' is added to the power consumption, for example by using an internal clock, by shuffling operations or by masking the internal computations that could be predicted by the attacker (Mangard et al., 2007), by using balanced data encoding or balanced place and route (Guilley et al., 2010), by using power filters or electromagnetic shields (Shamir, 2000), or by using software techniques called 'hiding' (Mangard et al., 2007).

2.2 Obfuscation as a general protection

We propose to make the distinction between obfuscation mechanisms that do not modify the behaviour of the target at runtime, thereafter called *static* obfuscation, and *dynamic* mechanisms which purpose is to modify, at runtime, the apparent behaviour of the target. Polymorphism, by providing to a secure component the capability to change form on a regular basis, is considered as a dynamic obfuscation mechanism.

2.2.1 Static obfuscation

Obfuscation of hardware is a common practice, because the attacker can gain a lot of knowledge about the target by physical inspection of its structure (Chakraborty and Bhunia, 2009) but we do not further consider this topic as it falls out of the scope of this paper.

Software obfuscation can involve modifications to the source code, to intermediate representations or to the binary code (Collberg et al., 1997) or encryption (Sander and Tschudin, 1998) in order to make it unintelligible or inaccessible to automatic analysis. At the theoretical level, it was demonstrated by Barak et al. that there is no general obfuscation technique that applies to any code function; in other words, there is no "universal obfuscating tool" (Barak et al., 2001). However, nothing in this theory forbids the existence of obfuscation techniques able to target the mostly used cryptographic algorithms such as DES, AES, RSA, ECC.

2.2.2 Dynamic obfuscation by polymorphism

Non-deterministic processors (May et al., 2001b) achieve what we call *polymorphic execution*, that is, the shuffled execution of a program from a static binary input residing in program memory. May et al. achieve dynamic instruction shuffling (May et al., 2001b) and random register renaming (May et al., 2001a). (Bayrak et al., 2012) describe an instruction shuffler: a dedicated IP inserted between the processor and the program memory (instruction cache).

Most (if not all) tools for program analysis and reverse engineering of software cannot handle programs that are dynamically modified at runtime (Madou et al., 2006). Hence, despite the theoretical work of (Barak et al., 2001), dynamically mutating code and polymorphic code are likely to make 'first step' attacks more challenging. (Madou et al., 2006) use runtime code rewriting scripts to restructure the contents of a procedure from binary programs under a scrambled form. However, this form of dynamic mutation is only effective against code analysis and not against physical attacks since the executed program always takes the same form.

To the best of our knowledge, only a few research works have studied the use of code polymorphism for security purposes. (Amarilli et al., 2011) propose to shuffle instructions and basic blocks of the program's control flow graph at runtime from a static version of the program. Their approach, applied to the shuffling of basic blocks only, is shown to increase the difficulty of DPA on AES. (Agosta et al., 2012) use runtime code generation (described by the authors as *code morphing*) to protect an implementation of AES against DPA. A runtime code generator is generated at static compilation time, embedded in the application binary code. The polymorphic production of code variants involves register renaming, instruction shuffling and the production of semantically equivalent code fragments. The overhead of code generation is significantly high since each new code generation is reported to execute in 90 ms ($11,970.10^6$ cycles at 133 MHz) on an ARM926 processor running GNU/Linux, in order to generate polymorphic versions corresponding to the 64 xor operations in the AES kernel. This approach is the closest to the approach that we envision in this paper, but as we will illustrate below, we expect our approach to better fit with the constraints of resource constrained embedded systems.

3 Code polymorphism for embedded systems

As explained in the previous section, the state of the art protections present the following limitations:

1. *General protections*, which make the code execution harder to interpret and so the knowledge about the circuit's functioning harder to recover, are a primary requirement in secure devices. However, state-of-the-art solutions present performance issues or are not applicable to secure devices because of resource constraints.
2. *Security in a device is incrementally built by adding standalone ad hoc protections.* Ad hoc countermeasures are often proven to be efficient when analysed independently, but are difficult to integrate in a device since each countermeasure comes with an additional performance overhead.

We postulate that code polymorphism is able to provide an efficient solution as a general protection against 'first step' attacks. Furthermore, 'second step' attacks, which rely on a precise behavioural (spatial and temporal) model of the target, are also more difficult to achieve if the target's behaviour is regularly modified. Indeed, the success of a fault attack depends on the temporal and spatial precision required to inject the fault at the right time during program execution, and on the right position on the chip; this should be more difficult in presence of polymorphism. Similarly, side channel attacks need to correlate a large number of activity traces to extract the target information; a polymorphic component is able to produce a larger set of different activity traces, and as a consequence, the attacker needs to gather a greater number of activity traces to achieve the attack.

In the work presented in this paper, our objective is to provide a realistic solution to achieve code polymorphism by introducing in a unique framework:

- Introduce alea during runtime code generation to produce polymorphic application components,
- Use semantic equivalences at the instruction level to produce different (but functionally equivalent) instances of code sequences
- Shuffle, at runtime, the machine instructions and randomise the mapping to physical registers
- Combine with hardware protections
- With limited memory consumption and fast code generation, so that it is applicable very small computing units such as secure elements

3.1 Sketching `deGoal`

`deGoal` is a framework for runtime code generation, whose initial motivation is the use of runtime code specialisation to improve program performance, e.g. execution time, energy consumption or memory footprint. In this section, we first sketch the characteristics of `deGoal` and then elaborate about its use in the context of security. (Couroussé et al., 2013; Aracil and Couroussé, 2013; Charles et al., 2014) further elaborate about the use of `deGoal` in contexts or application domains that originated its design.

In classical frameworks for runtime code generation such as interpreters and dynamic compilers, the aim is to provide a generic infrastructure for code generation, bounded by the syntactic and semantic definition of a programming language. The generality of such solutions comes at the expense of an important overhead in runtime code generation, both in terms

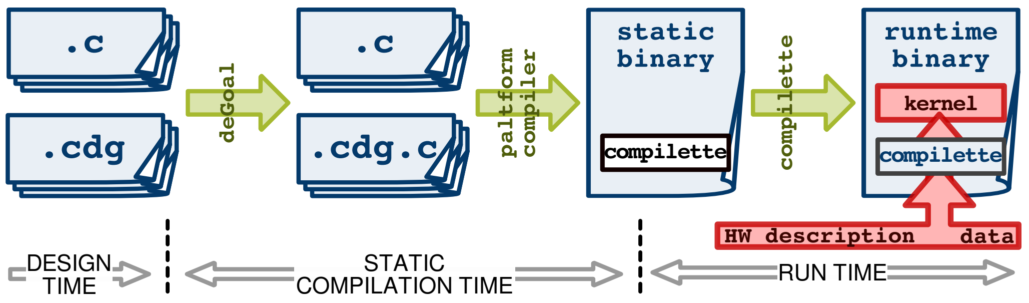

Figure 1: `deGoal` workflow: from the writing of application's source code to the execution of a kernel generated at runtime

of memory footprint and computing time and computing energy. In `deGoal`, a different approach is used: code segments (thereafter called *kernels*) are generated and tuned at runtime by ad hoc runtime code generators, called *compilettes*. Each compilette is specialised to produce the machine code of one kernel. Syntactic and semantic analyses are performed at the time of static compilation, and compilettes embed only the processing intelligence that is required for the runtime optimisations selected. As a consequence, compilettes offer very fast code generation (10 to 100 times faster than typical frameworks for runtime code interpretation or dynamic compilation), present a very low memory footprint, can run on very small microcontroller architectures such as 8/16-bit microcontrollers (Aracil and Couroussé, 2013), and are portable (Charles et al., 2014).

In COGITO, we re-target the original purpose of `deGoal` in order to focus on security aspects: we exploit the flexibility brought by `deGoal` for runtime code generation to achieve code polymorphism.

3.2 Application building, runtime code generation and execution

The building and the execution of an application using `deGoal` consists of the following steps as illustrated in Figure 1: writing the source code using a mix of `C` source code and of our dedicated `cdg` language (described below); compiling the binary code of the application and the binary code of compilettes using static tools; at runtime, generating the binary code of kernels by compilettes and in the end running the kernels.

3.3 The `Cdg` language

`Cdg` is an assembly-like DSL. From the programmer's perspective, this represents a major paradigm shift: `Cdg` expressions describe the instructions that will be *generated* at runtime instead of the instructions to be executed.

Compilettes are implemented by using a mix of ANSI `C` and `Cdg` instructions. The `C` language is used to describe the control part of the compilette that will drive code generation, while `Cdg` instructions perform code generation. The `Cdg` instruction set includes a variable length register set, classical arithmetic instructions, load and store instructions. From this high-level instruction set, compilettes map the `Cdg` instructions to machine instructions according to (1) the characteristics of the data to process, (2) the characteristics of the execution context at the time of code generation, (3) the hardware capabilities of the target processor, (4) execution time and/or energy consumption performance criteria. In all cases, code generation is fast, produces efficient code, and is applicable to low-resource embedded systems such as micro-controllers.

In the case of the work presented in this paper, we remove the dependency of code generation to the input data, and make code generation dependant of an alea to introduce random polymorphism.

3.4 Main properties of the code generation backend

Considering that `deGoal` targets constrained embedded systems, drastic architecture choices have been made so that code generation can be fast and memory lightweight. At runtime, a compilette performs in this order: register allocation, instruction selection and instruction scheduling.

Register allocation. In dynamic compilers, register allocation is usually performed *after* instruction scheduling: instruction selection and instruction scheduling are performed on a SSA form. The SSA form is later analysed and register allocation is performed, using techniques that provide a reduced runtime computational cost of the register allocation as compared to graph colouring usually used in static compilers (Kotzmann et al., 2008). On the contrary, in a compilette register allocation is done first, simply by using greedy algorithm. The idea is to lighten the pressure on instruction selection and instruction scheduling: if register allocation is done first, it becomes possible to perform instruction scheduling without intermediate representation[1]. This comes at the expense of a potential reduced code quality: currently, compilettes do not support register spilling.

Instruction selection. Instruction selection is performed at runtime once the runtime constants have been evaluated by the compilette. Instruction selection is done at the level of `Cdg` instructions: each `Cdg` instruction can be mapped to one or more machine instructions depending on the data input of the compilette and/or processor capabilities (e.g. availability of SIMD units).

Instruction scheduling. The generated machine instructions can be either directly written in program memory in order to fasten code generation, or pushed into an intermediate instruction buffer that is processed by an instruction scheduler. We plan to exploit this feature in this work to perform instruction shuffling during runtime code generation. A functional overview of instruction scheduling is illustrated in (Couroussé et al., 2013).

3.5 Implementation of code polymorphism with `deGoal`

We present now the features that we plan to implement in `deGoal` in order to achieve code polymorphism for security purposes.

Introduce alea during runtime code generation. Code polymorphism has to present a fair level of randomness. Each new code generation should produce a code sequence with a different binary code, while preserving the functionality of the original implementation. At the same time, the process of code generation must remain deterministic, considering a selected random input.

Random generation of semantically equivalent sequences. Our runtime code generation technique is able to involve all the optimisation techniques usually employed in compiler backends, where the purpose is to optimise the machine code generated according to the characteristics of the target. In our case, we leverage this level of variability to increase the number of code variants possibly generated. We consider randomly changing the following code generation parameters:

- use of semantic equivalences; for example replacing a multiplication operation with a power of two with a left shift operation (Agosta et al., 2012).

  Semantic equivalence can involve more complex patterns than a one-to-one relationship between equivalent operations, hence they can change the number of machuine instructions generated.
- insertion of dummy operations (Mangard et al., 2007), that is, operations intended to vary the execution time and the power trace of the target, but which do not impact the result of the processing.
- mapping to physical registers,
- instruction scheduling,

[1]or at least with a minimalist, much lighter intermediate representation

Ability to combine with hardware countermeasures. Use of dedicated hardware support or specialised instructions was a primary concern in the design of the `Cdg` language: hardware protections or hardware IPs (e.g. AES) are easily integrated in the implementation of a compilette. Furthermore, `Cdg` being a low-level programming language, it is easy to integrate software protections at the instruction level.

Limited memory consumption. `deGoal` compilettes embed only the necessary data to generate the instructions (and algorithmic equivalences in the case of this project) required for the target polymorphic code. For example, we achieved runtime code generation with `deGoal` on the evaluation board of the MSP430 micro-controller, which includes only 512 bytes of RAM (Aracil and Couroussé, 2013).

Fast runtime code generation. Using `deGoal`, the current speed of code generation ranges from 10 to 100 cycles per instruction generated in an implementation that is not targeting secure devices (Couroussé et al., 2013). Even with the addition of extra processing to achieve the insertion of randomisation at various levels, we expect the speed of code generation to remain far above the state of the art.

Portability to very small processors and secure elements is achieved thanks to the small footprint of compilettes and the small amount of processing they require to achieve code generation.

4 Conclusion

In this paper we advocate the use of code polymorphism as a means to secure the sensitive components of embedded applications such as cryptographic primitives. We assume that code polymorphism can bring improvement to several security issues at the same time. Firstly, it is de facto a mean to obfuscation, hence making reverse engineering more difficult to achieve. Moreover, considering that physical attacks are strongly related to the temporal and spatial behaviour of the target, regularly modifying the target's behaviour will raise the level of security of a component against physical attacks.

In the scope of the COGITO project, we plan to implement code polymorphism in secure components thanks to `deGoal`, a framework for runtime code generation that is applicable to embedded devices even with limited memory and computing resources. We have presented the various ways to leverage polymorphism, and illustrated them with a simple case. A large body of the work planned in this project will also consist in the analysis of the potential flaws that runtime code generation could bring to secure devices.

ACKNOWLEDGEMENTS

This work was partially funded by the French National Research Agency (ANR) as part of the program Digital Engineering and Security (INS-2013), under grant agreement ANR-13-INSE-0006-01.

REFERENCES

Agosta, G., Barenghi, A., and Pelosi, G. (2012). A code morphing methodology to automate power analysis countermeasures. In *DAC*, pages 77–82. ACM.

Amarilli, A., Müller, S., Naccache, D., Page, D., Rauzy, P., and Tunstall, M. (2011). Can Code Polymorphism Limit Information Leakage? In *WISTP*, LNCS 6633, pages 1–21.

Aracil, C. and Couroussé, D. (2013). Software acceleration of floating-point multiplication using runtime code generation. In *ICEAC*, pages 18–23.

Arora, D., Ravi, S., Raghunathan, A., and Jhaals, N. K. (2005). Secure Embedded Processing through Hardware-Assisted Run-Time Monitoring. In *DATE*, pages 178–183.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2001). On the (im) possibility of obfuscating programs. In *CRYPTO*, pages 1–18. Springer.

Barbu, G., Thiebeauld, H., and Guerin, V. (2010). Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In *CARDIS*, volume 6035 of *LNCS*, pages 148–163. Springer.

Bayrak, A. G., Velickovic, N., Ienne, P., and Burleson, W. (2012). An architecture-independent instruction shuffler to protect against side-channel attacks. *ACM TACO*, 8(4):20:1–20:19.

Chakraborty, R. and Bhunia, S. (2009). Harpoon: An obfuscation-based soc design methodology for hardware protection. *TCAD*, 28(10):1493–1502.

Charles, H.-P., Couroussé, D., Lomller, V., Endo, F., and Gauguey, R. (2014). deGoal a Tool to Embed Dynamic Code Generators into Applications. In *Compiler Construction*, volume 8409 of *LNCS*, pages 107–112. Springer.

Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland.

Couroussé, D., Lomüller, V., and Charles, H.-P. (2013). *Introduction to Dynamic Code Generation – an Experiment with Matrix Multiplication for the STHORM Platform*, chapter 6, pages 103–124. Springer.

Dutertre, J.-M., Possamai Bastos, R., Potin, O., Flottes, M.-L., Rouzeyre, B., and Di Natale, G. (2013). Sensitivity tuning of a bulk built-in current sensor for optimal transient-fault detection. *Microelectronics Reliability*, 53(9):1320–1324.

Genkin, D., Shamir, A., and Tromer, E. (2013). RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. Cryptology ePrint Archive, Report 2013/857.

Guilley, S., Sauvage, L., Flament, F., Vong, V.-N., Hoogvorst, P., and Pacalet, R. (2010). Evaluation of power constant dual-rail logics countermeasures against DPA with design time security metrics. *IEEE Trans. Computers*, 59(9):1250–1263.

Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., and Cox, D. (2008). Design of the java hotspot client compiler for java 6. *ACM TACO*, 5(1):7:1–7:32.

Madou, M., Anckaert, B., Moseley, P., Debray, S., Sutter, B., and Bosschere, K. (2006). Software protection through dynamic code mutation. volume 3786 of *LNCS*, pages 194–206. Springer.

Mangard, S., Oswald, E., and Popp, T. (2007). *Power analysis attacks: Revealing the secrets of smart cards*. Springer.

May, D., Muller, H., and Smart, N. (2001a). Random Register Renaming to Foil DPA. In *CHES*, volume LNCS 2162, pages 28–38. Springer.

May, D., Muller, H. L., and Smart, N. P. (2001b). Nondeterministic processors. In *ACISP'01*, pages 115–129. Springer.

Moro, N., Heydemann, K., Encrenaz, E., and Robisson, B. (2014). Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, pages 1–12.

Novak, R. (2003). Side-channel attack on substitution blocks. volume 2846 of *LNCS*, pages 307–318. Springer.

Petroni, Jr., N. L. and Hicks, M. (2007). Automated detection of persistent kernel control-flow attacks. In *CCS*, pages 103–115. ACM.

Sander, T. and Tschudin, C. (1998). On software protection via function hiding. In *Information Hiding*, volume 1525 of *LNCS*, pages 111–123. Springer.

Shamir, A. (2000). Protecting smart cards from passive power analysis with detached power supplies. In *CHES*, LNCS, pages 71–77. Springer.

Zussa, L., Dehbaoui, A., Tobich, K., Dutertre, J.-M., Maurine, P., Guillaume-Sage, L., Clediere, J., and Tria, A. (2014). Efficiency of a glitch detector against electromagnetic fault injection. In *DATE*, pages 1–6.