All paths lead to Rome: Polymorphic Runtime Code Generation for Embedded Systems

Damien Couroussé
Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France
damien.courousse@cea.fr

Nicolas Belleville
Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France
nicolas.belleville@cea.fr

Hélène Le Boudier
IMT-Atlantique, Rennes, France
helene.le-boudier@imt-atlantique.fr

Thierno Barry
Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France
thierno.barry@cea.fr

Philippe Jaillon
École Nationale des Mines de Saint-Étienne, France
philippe.jaillon@emse.fr

Jean-Louis Lanet
LHS, INRIA Rennes, France
jean-louis.lanet@inria.fr

Bruno Robisson
CEA-Tech DPACA, Gardanne, France
bruno.robisson@cea.fr

Olivier Potin
École Nationale des Mines de Saint-Étienne, France
olivier.potin@emse.fr

Karine Heydemann
Sorbonne Université, CNRS, LIP6, F-75005, Paris, France
karine.heydemann@lip6.fr

CCS CONCEPTS
• Security and privacy → Side-channel analysis and countermeasures; Software security engineering; • Software and its engineering → Compilers;

ACM Reference Format:

1 PHYSICAL ATTACKS: A MAJOR CHALLENGE FOR EMBEDDED SYSTEMS

In the landscape of cybersecurity, a large field of research is dedicated to physical attacks since the publication of the first attacks in the early 1990s. Side-channel attacks can reveal the secret values processed in a circuit by observing physical quantities (power consumption, electro-magnetic emissions, execution time, etc.).

Physical attacks constitute an important threat against embedded systems; in particular, they are the most effective way to break implementations of cryptography. The Smart Cards industry is up with the design of countermeasures, and high security products embed a large set of hardware and software countermeasures. With the emergence of the Internet of Things, we observe a rapid increase of the number of communicating devices, which present various security needs, but also unequal levels of security [7]. Hence, we advocate for the design of tools to automate the application of counter-measures on a large scale of programs and processor architectures.

2 CODE POLYMORPHISM: BEHAVIOURAL VARIABILITY AS A PROTECTION PRINCIPLE

In the COGITO project, we focused on the use of runtime code generation to introduce behavioural variability in embedded systems. Indeed, behavioural variability is often used as a protection against physical attacks [6]. Security products embed hardware and software desynchronisation mechanisms to achieve variability in side-channels: for example clock jitters in hardware or dummy loops of random duration in software. We defined *code polymorphism* as the capacity of a program component to vary its observable behaviour, at runtime, without altering its functional properties. Code polymorphism can be considered as a *hiding* countermeasure: the information leakage, which is observable physical quantities during the secured computation, is hidden in the information noise produced by the behavioural variability generated by the polymorphism. However, code polymorphism alone does not remove information leakage as it would be the case with masking countermeasures.

We implemented code polymorphism with runtime code generation of machine binary instructions (Fig. 1): the polymorphic component is composed of (1) dedicated runtime code generators, specialised for the targeted component so that it presents a low memory footprint and a short code generation time, and (2) of *polymorphic instances* which are the many code variants produced by the polymorphic code generator at runtime. In order to produce many code variants of the same functional component, the runtime code generator is driven by a source of random data. The successive execution of many polymorphic instances, which are all functionally equivalent but composed of different series of machine instructions, will induce a strong variability in the observable behaviour of the polymorphic component.
4.3 Design of the toolchain

In the early stages of the project, discussions with industrial bodies and the transfer of a prototype implementation to all partners led us to consider issues related to the industrialisation and the usability of a toolchain for securing embedded software:

Determinism and reproducibility. Runtime code generation is driven by random data so that an attacker cannot predict the observable behaviour of a polymorphic program. However, knowing the series of random values used, the behaviour of the runtime code generator is completely reproducible. In particular, this feature is mandatory for debugging purposes.

Debug tools. Runtime code generation represents a paradigm shift for the software developer as compared to traditional approaches where programming languages describe static instructions that will be executed by a processor. In our case, the source code implementation describes the execution of a runtime code generator, that will furthermore produce many different instances of the target program.

Functional validation. A large body of research works in computer science is focused on validating that a program is functionally correct. To the best of our knowledge, these research works target the validation of static programs mostly. Polymorphic code generation represents an interesting challenge, since in this case it is necessary to demonstrate the validity of the whole class of polymorphic instances targeted by a polymorphic code generator. All the polymorphic code transformations involved in our toolchain have been designed to preserve the functional validity of the many polymorphic instances produced. Until now however, we could only conjecture about their correctness. Dedicated methods need to be designed to achieve this goal.

ACKNOWLEDGMENTS

The works presented in this paper were partially funded by ANR under grant agreements ANR-13-INSE-0006-01 (project COGITO) and ANR-15-CE39 (project PROSECCO).

REFERENCES