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

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

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

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

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

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

Hélène Le Bouder
IMT-Atlantique, Rennes, France
helene.le-bouder@imt-atlantique.fr

Jean-Louis Lanet
LHS, INRIA Rennes, France
jean-louis.lanet@inria.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:
Damien Couroussé, Thierno Barry, Bruno Robisson, Nicolas Belleville, Philippe Jaillon, Olivier Potin, Hélène Le Bouder, Jean-Louis Lanet, and Karine Heydemann. 2018. All paths lead to Rome: Polymorphic Runtime Code Generation for Embedded Systems. In *CS2 '18: Fifth Workshop on Cryptography and Security in Computing Systems, January 24, 2018, Manchester, United Kingdom.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3178291.3178296

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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CS2 '18, January 24, 2018, Manchester, United Kingdom
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6374-7/18/01.
https://doi.org/10.1145/3178291.3178296

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.

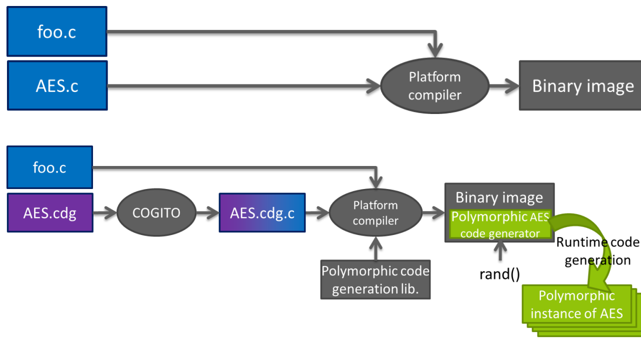

Figure 1: Overview of a reference compilation toolchain (top), and of a compilation toolchain for the application of code polymorphism with COGITO

3 MAIN PROJECT RESULTS

In our implementation of code polymorphism, each runtime code generator is specialised for a dedicated polymorphic program. At runtime, the generators perform code transformations involving register renaming, semantic variants, insertion of useless machine instructions, and instruction reordering. All those transformations are highly parametrisable according to performance and security constraints. Specializing the code generator is twofold: (1) it leverages fast runtime code generation and small memory footprints in order to minimise the overhead incurred by runtime code generation; (2) it prevents the injection of ill-formed or malicious programs via the code generator (e.g. JIT spraying attacks).

We have shown that it is possible to implement a fully polymorphic software AES on an embedded system with a 32-bit ARM Cortex-M core and only 8 kBytes of RAM, showing an increase of execution time from 2× to 20× [3]. The overheads are highly variable: they depend on the algorithmic nature of the original program to protect, the level of behavioural variability and polymorphic code transformations, and the frequency of code regeneration.

4 DISCUSSION

4.1 Resistance to side-channel attacks

Our work follows other research studies showing that code polymorphism is highly effective against side-channel attacks [1, 2, 4, 5]: the number of side-channel observations necessary to perform a successful attack is increased to the point that the attack is not longer tractable in practice. Furthermore, it was shown that polymorphism can reduce information leakage to an undetectable level according to standards of experimental evaluation such as the *t-test* [2]. However, to the best of our knowledge, the resistance of polymorphism was not investigated against the most recent side-channel attacks involving advanced signal processing or machine learning techniques.

4.2 Certification

All the products that embed cybersecurity solutions must be certified according to one or several certification standards before being sold on the market. In collaboration with the National Cybersecurity Agency of France (ANSSI), we have demonstrated that our implementation of code polymorphism is compatible with the current certification standards, in particular *Common Criteria*.

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

[1] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. 2012. A Code Morphing Methodology to Automate Power Analysis Countermeasures. In *DAC*. 77–82.
[2] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2015. The MEET Approach: Securing Cryptographic Embedded Software Against Side Channel Attacks. 34, 8 (2015), 1320–1333.
[3] Damien Couroussé, Thierno Barry, Bruno Robisson, Philippe Jaillon, Olivier Potin, and Jean-Louis Lanet. 2016. Runtime Code Polymorphism as a Protection Against Side Channel Attacks. Springer, 136–152.
[4] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity.. In *NDSS*. 8–11.
[5] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2013. Librando: Transparent Code Randomization for Just-in-Time Compilers. In *CCS (CCS '13)*. ACM, 993–1004.
[6] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Vol. 31. Springer-Verlag.
[7] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin OFlynn. 2017. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. IEEE, 195–212.