Journées 2022 du GT "Méthodes Formelles pour la Sécurité", GdR Sécurité Informatique

21-23 Mar 2022
Fréjus
France

Table of contents

A CompCert Backend with Symbolic Encryption

PAOLO TORRINI, INRIA, Grenoble, France
SYLVAIN BOULMÉ, Verimag, Grenoble, France

Binary encryption can be used to strengthen a verified compilation toolchain, in order to protect executable code from malware attacks. We present IntrinSec, a C backend that extends RISC-V with binary encryption and our work on its formalization and verification as a CompCert backend.

Attacks against computer systems can take advantage of software vulnerabilities such as buffer overflows in order to inject malicious code or divert control-flow. Protection against them typically involves ensuring integrity of executable code and stack data. Integrity can then be used to ensure higher-level properties of system behaviour, such as those represented by the control-flow directed graph (CFG) and non-interference. Various techniques have been introduced to ensure integrity, including mitigation tools, type-safe languages, the enforcement of CFG-based control-flow integrity [1] and encryption.

Binary code encryption can be used to safe-guard code and data integrity. Unlike other approaches, it generally requires the deployment of specialized hardware to execute the encrypted code. In [2, 4], integrity of C program execution by authenticated encryption of instructions is ensured by compilers that have been co-developed with RISC processors (the latter used as prototype for proprietary CEA hardware developed within the NanoTrust project). The encryption of binary programs is carried out at compile-time by the trusted compiler. Their decryption is achieved on the fly, at runtime, by the processor itself. The processor supports single-instruction decryption (it executes cyphertext by decrypting each instruction just before executing it).

1 INTRINSEC

The IntrinSec assembly, following the design of [4], extends RISC-V 32bits with additional registers and instructions, for control-flow monitoring (CFM), which appear in **blue** on Fig. 1. The compiler translates source code to encrypted binary code (EBC) after linking, producing cyphertext which is executable relying on single-instruction decryption. Encryption is based on stream cyphers (finite ones), each associated with a code block (cryptographic block) that has a single entry point. Instructions in the block are sequentially associated with CFM tokens (which are masks) [2]. The assembly code is instrumented at compile-time in order to update the CFM tokens accordingly to control-flow branch. A higher level of protection can be achieved by adding encryption of the whole program, associated with a stronger secret key, and by introducing data encryption, though these aspects will not be further discussed here.

Verified compilation ensures source-level behaviour preservation, i.e. that the target assembly code always behaves compatibly with the source code semantics. The CompCert C compiler [3] formally developed and verified in Coq is based on a chain of verified compilation passes between intermediate languages down to Asm. Each

```
int fact(int n){
  if (n <= 1) return 1;
  return n*fact(n-1);
}
```

```
  ecr.enter
fact:
  mv        x30, sp
  addi      sp, sp, -16
  sw        x30, 0(sp)
  sw        ra, 4(sp)
  ecr.sw    emr, 8(sp)
  sw        x8, 12(sp)
  mv        x8, ra0
  ecr.lui   emb,%hi(.L100)
  ecr.addi  emb, emb,%lo(.L100)
  addi      x31, x0, 1
  blt       x31, x8, .L100
```

```
  addi      ra0, x0, 1
  ecr.lui   emb,%hi(.L101)
  ecr.addi  emb, emb,%lo(.L101)
  j         .L101
.L100:
  addi      ra0, x8, -1
  ecr.lui   emb,%hi(fact)
  ecr.addi  emb, emb,%lo(fact)
  call      fact
  mul       ra0, x8, ra0
.L101:
  lw        x8, 12(sp)
  lw        ra, 4(sp)
  ecr.lw    emb, 8(sp)
  addi      sp, sp, 16
  jr        ra
```

Fig. 1. IntrinSec assembly produced by our version of CompCert

language is characterized in terms of executable semantics, and they all share a memory model which ensures separation between mutable data and code associated with functions. At the assembly level, a function is associated with a block and each instruction in the block with an offset. CompCert can target different assembly backends including RISC-V [5].

Here we discuss the formalization and verification of the IntrinSec CompCert 3.8 backend[1]. In order to deal with cryptographic tokens, we extend the RISC-V formal model with the specific instructions and registers, and we extend the memory state with a correspondingly modified notion of stack frame. This instrumented version of CompCert RISC-V constitutes the executable model of the IntrinSec backend. As CompCert verified compilation stops at the assembly code, the formal verification of our compiler can only consider an abstract model. We extend the IntrinSec executable model with an axiomatic model of link-time encryption and run-time decryption, basically corresponding to a symbolic encryption model of stream cyphers in Coq. Encryption is represented as a function that depends on a cryptographic block and an offset, returning the CFM mask of the corresponding instruction.

Our basic encryption model assumes that each function block, hence each function, is associated with a stream cypher. The first instruction in the block is associated with the initial mask of the stream (entry mask generated by the "`ecr.enter`" directive in the concrete assembly). Further instructions in the code are then sequentially associated with further masks in the stream. In order to decrypt an instruction, the processor needs to be given the correct mask: an incorrect mask is considered as the result of a control-flow

Authors' addresses: Paolo Torrini, INRIA, Grenoble, France, Paolo.Torrini@inria.fr; Sylvain Boulmé, Verimag, Grenoble, France, Sylvain.Boulme@univ-grenoble-alpes.fr.

[1] https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-intrinsec.

attack, and the execution aborts. The processor fetches this mask from dedicated mask registers: MSK_CNT (for the mask associated with the next instruction address, given by the program counter PC), **emr**—called MSK_RTN in our Coq model—(for the mask associated with the return address, given by RA), and **emb**—called MSK_BRN in Coq—(for the mask associated to the next branching address). On each non-branching instruction, MSK_CNT is implicitly updated (with the symbolic encryption). For jumps, the correct mask (stored in the memory at conventional locations for indirect jumps), must be loaded by the program to **emb** by means of the special "**ecr.**" instructions (see Fig. 1).

Asm programs in CompCert are obtained from a chain of intermediate languages inclusive of Mach and Linear [3]. This makes it possible to take advantage of the Mach semantics in the verification of control-flow properties, in particular with respect to function entry points. A minor revision of the semantics preservation proof between Mach and Linear is needed in connection with stackframe operations. A more significant revision is needed for the semantics preservation proof from Mach to Asm. The verification invariant needed for IntrinSec strengthens RISC-V source-level behaviour preservation with a symbolic encryption invariant, ensuring that an instruction can be executed only if the mask which becomes available for its decryption matches the one that has been used to encrypt it (expressed as a relation between PC and MSK_CNT). The proof consists in a refactoring of the analogous one for Mach and RISC-V, and it involves a significant revision of the notion of straightline code, which is essentially meant to capture code without jumps.

From the point of view of symbolic encryption, under the assumption that masks cannot be guessed and encryption cannot be broken, our model can guarantee code integrity, i.e. the fact that it is not possible to make the processor execute assembly code that has been altered or introduced by the attacker. This protection extends at least partially to control flow, especially with respect to function calls (i.e. the forward edges in the CFG). In the case of direct jumps, the destination address is protected by mask encryption. In the case of indirect ones, the mask to decrypt the address is associated to the function entry point. This is safe, under the assumption that the conventional location at which the mask is stored cannot be guessed. Moreover, under the assumption that the mask increment function cannot be guessed, this suffices to guarantee that jumping into the middle of a function is not possible. Concerning the return addresses (i.e. the backward edges in the CFG), the problem is more delicate as both the return address and the associated return mask are stored as data on the stack. Therefore, in order to protect the backward edges, data encryption is needed.

2 RESETTING CRYPTO-BLOCKS

The basic version of IntrinSec assumes that cryptographic blocks coincide with function blocks. This means that each stream cypher has at least the same size of the block it is associated with. However, this is a rather artificial constraint and may be undesirable when function blocks are very large. It seems then appropriate to introduce an independent notion of crytographic block. Operationally, this can be done by using a special label to mark the start of a new block, and a reset instruction to initialize the stream cypher. Although the semantics of reset seems easy, this instruction complicates the refactoring of the proofs, as the encryption function comes to depend on the whole function code (labels may trigger a reset), and this makes the definition of the encryption invariant more complex.

The CompCert inductive definition of "straightline code" is semantic (i.e. bounding the PC shift between two sequential instructions). This definition does not depend on any instruction set, and therefore can be used for different backends. In general, this notion is slightly different from a syntactical one which could be obtained by checking for jump instructions in the function code (a jump which advances the counter by one does not break the semantic definition). In IntrinSec, however, this notion has to be modified, as we need to keep into account the MSK_CNT update. With the reset instruction, this complicates refactoring. Thus, we switched to a syntactic notion of straightline code, excluding jump and reset.

3 PSEUDO-ASM

In order to mitigate the refactoring problem which may arise when we enrich the Asm back-end (as the reset example shows), we want to split two main aspects that are dealt with in the translation from Mach to Asm. One is the shift from the structural character of the stack representation in Mach to the memory-embedded one of Asm. The other one is the shift from the Mach instruction set to the Asm one. We then introduce an intermediate language, which we call PseudoAsm, that has the same instruction set as Mach but has PC and RA registers and a memory-embedded stack similar to the Asm one. Not only we can factor the translation from Mach to Asm into two distinct ones, one from Mach to PseudoAsm and the other one from PseudoAsm to Asm. We can also define an inverse translation from PseudoAsm to Mach, under a state match relation that restricts Mach states to those with a stack that can be faithfully embedded in memory. An analogous restriction can be introduced in the translations from Linear to Mach, and from Mach to PseudoAsm. The advantage of this approach (currently work in progress) is not only to achieve better modularity, but also to make it possible to express security properties which can be preserved down to PseudoAsm, thus localizing their possible break-down to the shift from the Mach instruction set to the Asm one.

ACKNOWLEDGMENTS

This work has been partially supported by the IRT Nanoelec (ANR-10-AIRT-05), funded by the French national program "Investissement d'Avenir".

We also wish to thank Olivier Savry, Thomas Hiscock, Marie-Laure Potet and David Monniaux for their helpful collaboration.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. 2005. A Theory of Secure Control Flow. In *ICFEM (LNCS)*, Vol. 3785. Springer, 111–124.

[2] T. Hiscock, O. Savry, and L. Goubin. 2019. Lightweight instruction-level encryption for embedded processors using stream ciphers. *Microprocessors and Microsystems* 64 (2019), 43–52.

[3] X. Leroy, S. Blazy, Z. Dargaye, Jourdan J. H., M. Schmidt, B. Schommer, and J. B. Tristan. 2020. The CompCert C Compiler, Version 3.8. http://compcert.inria.fr/compcert-C.html

[4] O. Savry, M. El-Majihi, and T. Hiscock. 2020. Confidaent: Control FLow protection with Instruction and Data Authenticated Encryption. In *DSD 2020*. IEEE, 246–253.

[5] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi. 2014. *The RISC-V instruction set manual (TR EECS-2014-54)*. Univ. of Calif. Vol. 26.

A Formalization of the Proof of Correctness of a Number-Theoretic Transform in the Context of the Hakyber Cryptographic Primitive

Antoine Séré
Institut Polytechnique de Paris
antoine.sere@polytechnique.edu

Pierre-Yves Strub
Institut Polytechnique de Paris
pierre-yves.strub@polytechnique.edu

Context

Kyber[4] is a key-encapsulation mechanism (KEM) based on the hardness of the Module-LWE problem. It is part of the package CRYSTALS, submitted to the NIST Post-Quantum Cryptography project, and is currently a round 3 finalist in it's category.

A formally verified implementation of Kyber, called Hakyber, is currently in development. Hakyber is written in Jasmin[1], and formally verified in EasyCrypt[3].

EasyCrypt has already been used to prove the correction of an implementation of ChaCha20-Poly1305[2] in Jasmin. This proof consists in first proving the correction of a reference implementation of Poly1305, and then by *game-hopping*: proving the equivalence between gradually more optimized implementations.

Hakyber

Hakyber works over the base ring $R_q = \mathbb{Z}_{3329}[X]/(X^{256}+1)$. Elements of this ring are represented as arrays containing the Montgomery representation of their coefficients, stored in reversed bit order.

The multiplication is implemented using a variant of the number-theoretic transform (NTT). This implementation contains several optimizations rendered possible by the representation of R_q. Notably the final value of one of the loop indexes is reused outside of the loop body.

The correctness of the implementation of the NTT in Hakyber is thus an important part of the correction proof included in Hakyber. Hakyber also is formally proven in EasyCrypt to be IND-CPA and IND-CCA2 secure.

Contributions

We showed the correction of the NTT implementation in EasyCrypt using *game-hopping*, notably by proving the equivalence of an naive implementation with another working on the reverse bit order, and between the most optimized version, and one less optimized that does not reuse the loop indexes outside their associated loops.

The proof we produced involves more complex mathematical objects than the proof of correction of Poly1305, namely quotients of polynomials over a finite field. It shows an optimized NTT implementation corresponds to the NTT in R_q, and that the multiplication in R_q can be defined using the NTT by the formula: $\mathrm{NTT}^{-1}(\mathrm{NTT}(f) \circ \mathrm{NTT}(g)) = f \times g$.

We also added to the EasyCrypt standard library relevant lemmas to reason about reverse bit order. We also added lemmas to reason about particular forms of while loops commonly used, namely *for* loops, where the integer variables used in the while condition are only changed by the end of the loop body, by adding, multiplying or dividing by a constant. These lemmas allow reasoning easily about these special cases in EasyCrypt's HL, pHL and pRHL logics.

We also started to extend EasyCrypt with type classes, which would allow to more easily manipulate complex mathematical objects in EasyCrypt in other proofs.

References

[1] José Bacelar Almeida et al. "Jasmin: High-Assurance and High-Speed Cryptography". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* Ed. by Bhavani M. Thuraisingham et al. ACM, 2017, pp. 1807–1823. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134078. URL: https://doi.org/10.1145/3133956.3134078.

[2] José Bacelar Almeida et al. "The Last Mile: High-Assurance and High-Speed Cryptographic Implementations". In: *2020 IEEE Symposium on Security and Privacy (SP).* 2020, pp. 965–982. DOI: 10.1109/SP40000.2020.00028.

[3] Gilles Barthe et al. "EasyCrypt: A Tutorial". In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures.* Ed. by Alessandro Aldini, Javier López, and Fabio Martinelli. Vol. 8604. Lecture Notes in Computer Science. Springer, 2013, pp. 146–166. ISBN: 978-3-319-10081-4. DOI: 10.1007/978-3-319-10082-1_6. URL: https://doi.org/10.1007/978-3-319-10082-1_6.

[4] Joppe Bos et al. *CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM.* Cryptology ePrint Archive, Report 2017/634. https://ia.cr/2017/634. 2017.

Information flow is a class of security properties that characterizes the way information flows between different entities of a system. Opacity is one of these properties that offers a general framework allowing to specify a wide range of security properties and it characterizes the ability of a system to keep a secret information hidden from an attacker.

Considering a predicate on the system behavior, this predicate is said to be **opaque** if for every behavior satisfying the considered predicate, there is (at least) another behavior, not satisfying the predicate and that both of them are indistinguishable by the attacker .

Enforcing the opacity property has been studied within the framework of Supervisory Control Theory (SCT). The objective is to synthesize a supervisor that restricts the behavior of the system by disabling events that lead to secret divulgation and hence, opacity leakage. A key concept in Supervisory Control is **permissiveness**. Indeed, a supervisor has to enable the maximal number of events and is hence called the maximal supervisor.

We assume that we have a Discrete Event System G, that is partially observed by an attacker. Our work investigates the problem of enforcing the opacity of G from the Supervisory Control perspective. We suggest a novel methodology to synthesize a **maximal supervisor** that restricts the behavior of the considered system in the general case without any hypothesis on the relationship between the attacker and the supervisor's observations. Moreover, the presented approach has been implemented in C/C++ providing a simple framework to verify and enforce the opacity property.

Verifying the opacity of a system requires exploring its state space which leads to the combinatorial explosion. Instead of exploring all the states of a system, we regroup them into groups of states called "aggregates" and are encoded using Binary Decision Diagram techniques. The obtained graph is called Symbolic Observation Graph (SOG) and has been used to verify the opacity of Discrete Event Systems. The SOG is built based on the observation of both the attacker and the supervisor. The supervisor is then computed on-the-fly during the construction of this graph.

In our work, we propose an algorithm based on an on-the-fly construction of a new version of the SOG called Hyper Symbolic Observation Graph (HSOG, see Figure 1) where **nodes** [*super aggregates*, represented by red rectangles in Fig. 1] are sets of aggregates (not single states), **actions** (linking two aggregates) are observed by the supervisor only and **arcs** (linking two super aggregates e.g. the event {*a*} in Fig. 1) are labeled with actions observed by the attacker. This graph allows us to represent the state space in a condensed manner which alleviates the explosion state problem.

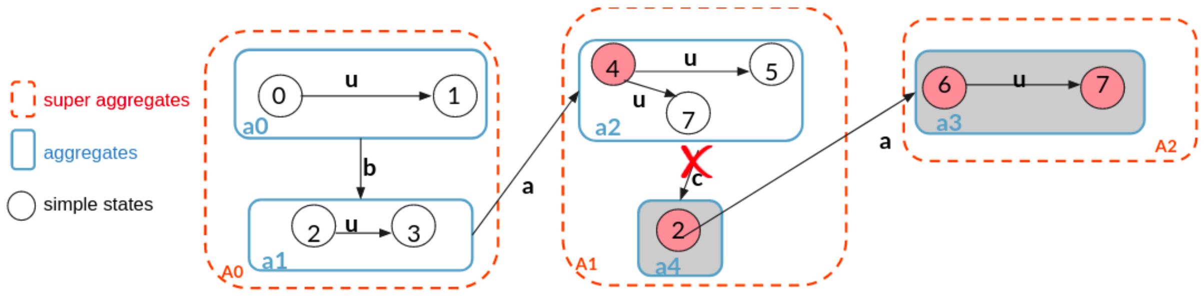

Figure 1

A comprehensive security analysis of the Belenios protocol and more

Alexandre Debant[*]

Université de Lorraine, LORIA, INRIA, CNRS, Nancy, France

Abstract

This work presents the results of a comprehensive security analysis of the e-voting protocol Belenios including verifiability and vote secrecy. In particular, it presents an unknown attack against vote secrecy when considering scenarios with multiple elections. In addition, it introduces a new and more practical fix to an already known attack against verifiability. Finally, it presents ongoing works about improving Belenios to ensure cast-as-intended and how this property can be proved using an existing verification tool such as ProVerif.

1 Context

Electronic voting protocols aim to achieve similar security guarantees than traditional, on-site, paper-based voting. Namely, the two main security objectives are ballot privacy (no one knows my vote) and verifiability (a voter can check that her vote has been counted, possible assisted by auditors). As for security protocols in general, designing protocols that are correct is difficult and many flaws are detected *a posteriori*. Due to the criticity of e-voting, the state-of-the-art practice consists in rigorously proving security. Actually, in Switzerland, this is even a legal requirement for e-voting : "a cryptographic and a symbolic proof must be provided" before a deployment.

2 Contributions

In this work, we present a security analysis of the Belenios protocol which claims to satisfy both vote secrecy and verifiability. Our contributions are as follows:

- first we conducted a security analysis with the will to model the most realistic scenarios possible. In particular, we decided to consider scenarios with multiple elections run in parallel or in sequence. We discovered a serious vote secrecy attack which allows an attacker to learn the vote of any targetted voter. The attack was possible because an attacker was able to replay a ballot from an election to another. We proposed and proved correct a fix which has already been implemented.
- second, we developed a comprehensive analysis of the verifiability property. In particular, we studied the security of the protocol depending on the assumptions that apply to the voters: the voter verifies her vote at the end of the election (which is not realistic...), or the voter verifies her vote just after having sent it (more realistic). In this last scenario, Belenios fails to ensure verifiability. We thus proposed a fix that could be implemented in practice based on the use counters. To prove its correctness we took advantage of the new version 2.3 of ProVerif which natively supports natural numbers.

[*]These works have been conducted in collaboration with Véronique Cortier

https://www.bk.admin.ch/dam/bk/en/dokumente/pore/Annex_of_the_Federal_Chancellery_Ordinance_on_Electronic_Voting_V2.0_July_2018.pdf

https://www.belenios.org/

3 Ongoing works

After having formally and comprehensively studied the security of Belenios, we are now working on how Belenios can be improved to remove the trust assumption on the voting device, i.e. the device used by the voter to create her electronic ballot. More precisely, we want to extend the Belenios protocol to ensure *cast-as-intended*, i.e. ensure that a voter can be convinced that her ballot actually contains her intended vote. In other words, the (possibly corrupted) voter's device should not be able tp modify voter's vote when creating the electronic ballot.

The solution we investigate relies on random audits performed by the voter. More precisely, the device must create three ciphertexts: $c = \{v\}_{pk}$, $c_1 = \{x\}_{pk}$, and $c_2 = \{v + x\}_{pk}$ where v is the vote of the voter and x is a random number chosen by the voter, and a proof π ensuring that the plaintext of c_2 is equal to the sum of the plaintexts of c and c_1. Once these four elements are received by the server, the voter is requested to chose $b \in \{1, 2\}$ and the device must open the corresponding ciphertext c_b. Since the device does not know which ciphertext will be audited in advance it cannot cheat on their values and thus on the value of the plaintext of c. More precisely, a device may cheat but it will be detected with probability $1/2$.

Unfortunately, existing verification tools such as ProVerif or Taramin cannot verify such probabilistic properties. We are thus developing techniques to get rid of these probabilities by proving sufficient non-probabilistic conditions which allow us to conclude that the property holds with probability $1/2$.

Analyse de contre-mesures dans le cadre d'attaques multiples

Etienne Boespflug

VERIMAG
University of Grenoble Alpes (UGA)
Grenoble, France

La sécurité des composants sensibles tels que les cartes à puce est une problématique majeure de nos jours. Les attaquants actifs nécessitent de raisonner à propos de modèles d'attaquants puissants capables de manipuler l'exécution et l'environnement du programme.

Les attaques par injection de fautes [BDL97, BBKN12] en sont un représentant populaire et utilisent des perturbations physiques (impulsions électromagnétiques [PTL+11], rayons de lumière focalisée [CMD+19], manipulation de la fréquence d'horloge [ZDC+12] par exemple) pour obtenir un avantage. Des techniques d'injections logicielles telles que Rowhammer [KDK+14] ont été développées. De façon similaire, les *insiders attacks* [PHN06] considèrent des attaquants ayant un accès sur la machine sur laquelle s'exécute le programme visé.

De nombreuses contre-mesures logicielles ont été proposées afin de détecter ou de corriger les fautes. Ces contre-mesures peuvent prendre des formes variées, et cette présentation s'intéresse aux contre-mesures basées sur les tests (*test-based countermeasures*) qui sont composées de portions de code appelées *détecteurs* visant à vérifier l'état du programme à des points précis afin de détecter, et éventuellement de corriger, une attaque.

Dans le cadre d'attaques multiples, une première attaque peut neutraliser une contre-mesure pour permettre à une seconde attaque d'atteindre son objectif. Cela implique que les détecteurs peuvent être contournés. Dans ce contexte, la combinaison des chemins d'attaque croît rapidement et il devient d'autant plus nécessaire de disposer d'outils et de méthodes automatiques pour évaluer la robustesse d'un programme et de ses contre-mesures, à la fois lors des processus de conception et d'évaluation d'un programme.

Nous proposons ici une méthodologie permettant de déterminer si certains détecteurs sont superflus pour un programme et un modèle d'attaquant donné. Nous proposons aussi une formalisation de cette méthode ainsi qu'une implémentation sur l'outil Lazart [PMPD14] reposant sur l'exécution symbolique.

La méthodologie, présentée à [BEPM20] prend en entrée un ensemble de traces d'exécution d'un programme contenant des détecteurs et un modèle d'attaquant et l'objectif d'attaque. Celle-ci s'articule en trois étapes:

- Classification des détecteurs (nécessaire, répétitif, inactif).
- Sélection des détecteurs en déterminant les ensembles de détecteurs minimaux sans introduire de nouvelles attaques.
- Génération du programme P' protégé.

La figure 1 décrit les résultats de notre méthode pour différents programmes d'exemples issus de la collection FISSC [DPP+16]: *verify_pin* (VP), *firmware updater* (FU), l'algorithme AES et l'exemple *GetChallenge* (GC). Trois contre-mesures sont étudiées: duplication de tests (TD), SecSwift CF (SSCF) [dF19] et LBH [LHB14]. La première colonne indique le programme protégé considéré et la seconde correspond au nombre total de détecteurs présents dans ce programme. Les colonnes suivantes indiquent le nombre de détecteurs retiré par notre approche pour chaque limite d'attaque (fautes).

Ces travaux sont réalisés dans le cadre de ma thèse intitulée "Outils pour l'analyse de code pour la sécurité des composants de confiance" effectuée sous la direction de Marie-Laure Potet et co-encadrée par Cristian Ene et Laurent Mounier. Ces travaux sont supportés par SECURIOT-2-AAP FUI 23, par l'Agence Nationale de Recherche dans le cadre du programme "Investissements d'avenir" (ANR-15-IDEX-02) et par le LabEx PERSYVAL Lab (ANR-11-LABX-0025-01).

Table 1: Taux de détecteur retiré pour chaque programme

Program	Total de détecteur	1 faute	2 fautes	3 fautes
VP + TD	11	72%	63%	18%
VP + SSCF	13	92%	76%	23%
VP + LBH	31	93%	93%	32%
FU + TD	14	0%	0%	0%
FU + SSCF	24	12%	12%	8%
GC + TD	39	37%	34%	34%
GC + SSCF	38	57%	28%	28%
AES C + TD	8	50%	50%	0%
AES C + SSCF	13	76%	61%	38%

References

[BBKN12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[BEPM20] Etienne Boespflug, Cristian Ene, Marie-Laure Potet, and Laurent Mounier. Countermeasures optimization in multiple fault-injection context. In *2020 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 26–34. FDTC, 2020.

[CMD$^+$19] Brice Colombier, Alexandre Menu, Jean-Max DUTERTRE, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. Laser-induced Single-bit Faults in Flash Memory: Instructions Corruption on a 32-bit Microcontroller. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–10, McLean, United States, May 2019. IEEE.

[dF19] François de Ferrière. A compiler approach to cyber-security. 2019 European LLVM developers' meeting, 2019.

[DPP$^+$16] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. FISSC: A Fault Injection and Simulation Secure Collection. In *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, pages 3–11, 2016.

[KDK$^+$14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[LHB14] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card C codes. In *Pr. of the 19th European Symposium on Research in Computer Security, ESORICS 2014*, pages 200–218, 2014.

[PHN06] Christian W Probst, René Rydhof Hansen, and Flemming Nielson. Where can an insider attack? In *International Workshop on Formal Aspects in Security and Trust*, pages 127–142. Springer, 2006.

[PMPD14] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014*, pages 213–222. IEEE, 2014.

[PTL$^+$11] François Poucheret, Karim Tobich, Mathieu Lisarty, L Chusseaux, B Robissonx, and Philippe Maurine. Local and direct em injection of power into cmos integrated circuits. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 100–104. IEEE, 2011.

[ZDC$^+$12] Loic Zussa, Jean-Max Dutertre, Jessy Clédiere, Bruno Robisson, Assia Tria, et al. Investigation of timing constraints violation as a fault injection means. In *27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France*, pages 1–6. Citeseer, 2012.

Assessing bug replicability for more security-minded bug finding

Guillaume Girol

Most problems in software verification are encoded as reachability queries of some undesired condition, for example a bug. We then have the guarantee that there is at least one program input triggering every reported bug. However, this tells us little of their security impact. Notably, can this bug be triggered by an attacker, or does it only happen in unreproducible initial conditions?

To be able to distinguish these cases, we introduce a new property called *robust reachability* [2] which refines the standard notion of reachability in order to take replicability into account. A bug is robustly reachable if a *controlled input* can make it so the bug is reached whatever the value of *uncontrolled input*. Robust reachability is better suited than standard reachability in many realistic situations related to security (e.g., criticality assessment or bug prioritization) or software engineering (e.g., replicable test suites and flakiness). We propose a formal treatment of the concept, and we revisit existing symbolic bug finding methods through this new lens. Remarkably, robust reachability allows differentiating bounded model checking from symbolic execution while they have the same deductive power in the standard case. Furthermore, we propose the first symbolic verifier dedicated to robust reachability: we use it for criticality assessment of 5 existing vulnerabilities, and compare it with standard symbolic execution.

The main short-coming of robust reachability is that it dismisses reports which can only be replicated 99% of the time, rather than 100%. We present preliminary results towards a more quantitative approach, putting a formal definition behind this value of 99%. Computing it looks similar but is distinct from standard model counting. We developed new algorithms inspired by d-DNNF-based model counters [1] to solve this new problem. This yields promising results for a form of security-minded bug triage.

1

References

[1] Adnan Darwiche. On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision. *Journal of Applied Non-Classical Logics*, 11:1–2, 2000.

[2] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 669–693, Cham, 2021. Springer International Publishing.

CV2EC: Getting the Best of Two Worlds

Bruno Blanchet[1], Pierre Boutry[2], Christian Doczkal[2,3], Benjamin Grégoire[2], and Pierre-Yves Strub[4]

[1]Inria
[2]Université Côte d'Azur, Inria
[3]Max Planck Institute for Security and Privacy
[4]École Polytechnique

The verification of cryptographic schemes - from the protocol level down to the correct implementation of the cryptographic primitives - is a challenging task. This has led to the development of a number of verification tools for cryptographic properties. These tools differ significantly as it comes to the properties they can express and reason about as well as the level of automation they provide. For instance, the CryptoVerif (CV) tool is primarily suited for verification at the protocol level and provides a high degree of automation. EasyCrypt (EC) on the other hand provides the expressive power to do verification at the protocol level and also reason about the correctness of cryptographic primitives. This comes at the cost of making things explicit that are implicit in CV and providing less automation.

We have developed a translation from CV to EC that allows cryptographic assumptions that cannot be proved in CV to be translated to EC and proved there. In fact, part of the high degree of automation in CV relies on stating cryptographic assumptions in a nonstandard form that allows the system to easily use them. In a similar way, the declaration of distributions in CV is very abstract, so notions like statistical distance can be stated (assumed) but can not be proved. By translating these assumptions to EC, we can reduce these non-standard assumptions to statements that more closely match what one would find in a paper proof.

More concretely, games in CV are simply collections of oracles, with the adversary and the part of the game that calls the adversary left implicit. As a consequence, CV cannot express any relationship between games where the signatures of the oracles (including the number of allowed calls) are different, making hybrid arguments almost impossible. In EC, adversaries and the outer part of the game are made explicit, allowing arbitrary reduction proofs.

As an example, consider a game that provides oracle access to an encryption oracle that can be queried N times. In CV, this looks as follows:

```
foreach i <= N do r <-R encseed;
  Oenc(m:plaintext) := return(enc(m, k, r))
```

and provides N (indexed) copies of the single-call encryption oracle `Oenc`. In EC, this would typically be represented as a procedure (in some module)

```
proc enc (m:plaintext) = { ...
  r <$ encseed; return enc(m, k, r); }
```

that can be called multiple times.

The main difficulties when translating from CV to EC are (a) that CV implicitly logs all variable assignments (e.g., the CV code above introduces an implicit array `r` of length N storing all encryption seeds) and (b) that the semantics of CV allows the adversary to trigger the sampling of `r` without calling `Oenc` at the same time. Both of these are solved by turning the replication index (e.g., `i`) into an argument to the procedure, maintaining maps that store all the necessary values, and providing all randomness through random oracles that become parameters of the game in EC. Thus, the inital example from CV is translated to EC roughly as follows, where `Oencseed.get` is a random oracle:

```
proc enc (i: int, m:plaintext) = { ...
  if (1 <= i <= N) {
    m_enc.[i] <- m; r <@ Oencseed.get(i);
    c.[i] <- enc(m, k, r); return c.[i]; }
```

To demonstrate the usefulness of the approach, we are working on a number of case studies. All of these involve some kind of hybrid arguments, given that such arguments are virtually impossible in CV:

- The reduction of the N query "real/ideal" formulation of the IND-CCA2 game in CV to the standard single-challenge formulation. (done)
- The reduction from the N participant games (e.g. insider or outsider adversaries) for authenticated KEMs to 1 or 2 participant games. (in progress)
- The reduction of the N query formulation of the Computational/Gap Diffie-Hellman (CDH/GDH) games in CV to the standard, single-query formulation. The obtained bounds are better than what can be obtained by a direct hybrid argument. (almost done)

The arguments for CDH/GDH require a series of changes to the way random values are sampled. To facilitate these arguments, we developed a new statistical-distance library for EC, allowing to bound the advantage of an adversary in terms of the statistical distance between the distributions used in the games.

Efficient Symbolic Algorithms for Software Verification Against Fault Attacks

Soline Ducousso*, Sébastien Bardin* and Marie-Laure Potet†

*Univ. Paris-Saclay, CEA, List, Saclay, France
†Univ. Grenoble Alpes, VERIMAG, Grenoble, France

soline.ducousso@cea.fr, sebastien.bardin@cea.fr, marie-laure.potet@univ-grenoble-alpes.fr

Extended abstract

Major works have delved into program analysis over the last decades, leveraging techniques such as symbolic execution [1], static analysis [2], abstract interpretation [3], or bounded model checking [4], to hunt for bugs and vulnerabilities, or even prove their absence [5], [6]. As bugs are the entry point of attacks, removing them is the first step towards software security. However, an attacker is not limited to crafting smart "input of death", but may as well take advantage of other attack vectors, such as (physical) hardware fault injections [7], micro-architectural attacks [8], [9] or a combination of any of the above.

While most program analysis tools for security only look for bugs, some recent works attempt to analyse a program taking into account attacker-induced faults. Yet, such state-of-the-art methods face scaling issues. Mutant generation [10] creates a new modified program for each faulted path. While other approaches [11]–[17], fork at each possible fault location. All the methods present an inability to scale, due to the induced path explosion problem when analysing a program with all possible fault locations – even more in a multi-fault context.

In this talk, we will report about our ongoing efforts toward providing efficient software verification techniques able to take the attacker into account. Especially, we introduce the Forkless technique which modifies the analysed expressions and instructions to embed the fault into them and avoid forking. We then use symbolic execution to compute the augmented path predicate and relegate finding the useful fault locations and values to an SMT solver. We design an efficient algorithm able to handle arbitrary data faults, variable reset and test inversion, which are amongst the most common fault models and encompass a wide range of attack situations. We implemented our technique inside BINSEC [18], a symbolic execution engine for binary analysis, and we evaluate it across benchmarks from the literature. We obtain an average speed-up of 800% compared with the baseline, exploring on average around 7 times less paths. We also report on a few security scenario we are investigating with this approach.

References

[1] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 2013.

[2] Facebook. Infer static analyzer. https://fbinfer.com/.

[3] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 2003.

[4] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 2001.

[5] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *Programming Languages and Systems*, 2005.

[6] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Form. Asp. Comput.*, 2015.

[7] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *VLSI*, 2013.

[8] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *TCAD*, 2019.

[9] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *SP*, 2019.

[10] Pablo Rauzy and Sylvain Guilley. A formal proof of countermeasures against fault injection attacks on crt-rsa. *Journal of Cryptographic Engineering*, 2014.

[11] Daniel Larsson and Reiner Hähnle. Symbolic fault injection. In *VERIFY*, 2007.

[12] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *ICST*, 2014.

[13] Guilhem Lacombe, David Féliot, Etienne Boespflug, , and Marie-Laure Potet. Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. *PROOFS*, 2021.

[14] Hoang M Le, Vladimir Herdt, Daniel Große, and Rolf Drechsler. Resilience evaluation via symbolic fault injection on intermediate code. In *DATE*, 2018.

[15] Jean-Baptiste Bréjon, Karine Heydemann, Emmanuelle Encrenaz, Quentin Meunier, and Son-Tuan Vu. Fault attack vulnerability assessment of binary code. In *CS2*, 2019.

[16] Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. Symplfied: Symbolic program-level fault injection and error detection framework. In *DSN*, 2008.

[17] Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, and Axel Legay. An automated formal process for detecting fault injection vulnerabilities in binaries and case study on present. In *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017.

[18] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *SANER*, 2016.

End-to-end enforcement of fine-grained "cryptographic" constant-time policies

Basavesh Ammanaghatta Shivakumar[1], Gilles Barthe[2], Benjamin Grégoire[3], Vincent Laporte[4], and Swarn Priya[3]

[1]MPI-SP,Germany
[2]MPI-SP,Germany IMDEA Software Institute, Spain
[3]Université Côte d'Azur, Inria, Sophia Antipolis, France
[4]Université de Lorraine, CNRS, Inria, LORIA

Timing attacks are a class of side-channel attacks in which attackers can learn about secret and otherwise protected values by measuring the execution time of programs operating on these values. A pragmatic approach used by crypto developers to minimize timing attacks is to reason about leakage (or absence thereof) in a commonly accepted leakage model. One such commonly accepted model is the baseline constant-time leakage model, which assumes that only memory accesses and control flow are leaked. While this leakage model is adequate for analyzing many attacks from the literature:

- it does not account for time-variable instructions, whose execution time depends on its operands;
- it excludes real-world code, which uses a weaker leakage model and consequently achieves higher performance.

We introduce a general class of fine-grained constant-time policies that supports both weaker and stronger leakage models and their combination.

- the baseline (BL) leakage model, where guards and memory addresses are leaked;
- the time-variable (TV) leakage model, where guards and memory addresses are leaked, and the modulo operation leaks the base-2 integer logarithm of its operands;
- the cache line model (CL), where guards and cache lines of memory addresses are leaked;
- the TV+CL model, combining the TV and CL models.

Then, we propose a two-step approach for enforcing fine-grained constant-time policies: first, prove that source programs are constant-time w.r.t. a fine-grained policy using relational Hoare logic, and then prove that compilation preserves constant-time w.r.t. a fine-grained policy.

As a motivating example, we study the function ssl3_cbc_copy_mac of CBC decoding in OpenSSL. And, our contributions are the following.

- a formalization of fine-grained constant-time policies;
- a generic proof that the Jasmin compiler preserves fine-grained constant-time policies. The proof takes the form of an instrumented correctness theorem and shows that leakage of assembly programs can be computed deterministically from leakage of source programs. The proof is fully mechanized in the Coq proof assistant;
- a method for reasoning about fine-grained constant-time policies using relational Hoare logic. The method is implemented for Jasmin programs and uses the EasyCrypt proof assistant as the backend;
- formal proofs that previously unverified cryptographic code is constant-time in a (non-baseline) leakage model. These proofs often involve nontrivial arithmetic reasoning that made them out of the scope of all prior tools.
- a theoretical attack against OpenSSL with 32-byte cache lines, and a formally verified fix.

The complete development is provided in supplementary material[1].

[1]`https://github.com/jasmin-lang/jasmin/tree/constant-time-op`

1

Equational Proofs for Distributed Cryptographic Protocols

Joshua Gancher, Kristina Sojakova, Leo Fan, Elaine Shi, Greg Morrisett

In the cryptographic literature, the most common framework for reasoning about
Protocols proven secure in UC are known to be concurrently composable. Informally, this means that if we take two protocols that are known to be sufficiently close and plug them into another protocol, then the two resulting larger protocols will also be sufficiently close. This form of composability is highly desirable since it allows us to reason about protocols *modularly*.

Unfortunately, a major drawback of the UC framework is the sheer amount of low-level arguments one has to carry out when constructing a formal proof: for instance, proving the correctness of a one-time pad protocol using Diffie-Hellman key exchange in EasyUC [3] required cca 18,000 lines of code. Directly scaling up to more complex protocols in very expressive UC-based frameworks such as EasyUC and CryptHOL [4] is therefore currently unmanageable: while these tools do make significant technical progress, they require the user to manually construct *bisimulations* – or relational invariants – and reason about them.
We propose IPDL (short for *Interactive Probabilistic Dependency Logic*), a language and proof system for conducting approximate equivalence proofs for distributed cryptographic protocols. The following features make IPDL particularly convenient to use for UC-style proofs:

Process Calculus Syntax In contrast to frameworks based on actors with mailboxes (such as UC [2], which uses Interactive Turing Machines for this purpose), IPDL uses a *process calculus* based on *channels* to model communication, similar to symbolic systems [1]. Each (non-input) channel in IPDL is assigned a unique *reaction* that fully specifies the behavior of the channel. This process calculus format gives a much finer-grained approach to protocol analysis that naturally supports modular reasoning. IPDL's process calculus expresses a broad range of cryptographic protocols, including multi-party computation protocols and n-party protocols, where n depends on a security parameter.

Approximate Equational Logic An IPDL program can be seen as a *system of equations* between channels over which one can perform equational reasoning. Our equational logic enables one to prove protocols secure in the simulation paradigm without resorting to any low-level bisimulation argument.

Semantics and Computational Soundness We prove our equational logic computationally sound using a novel operational semantics for IPDL, which uses a crucial *confluence* property to safely intermix the probability of cryptography and the nondeterminism of distributed programming.

Mechanization and Case Studies We have mechanized IPDL in Coq, and open sourced it at `https://github.com/ipdl/ipdl`. Using our mechanization, we have verified a number of case studies, which range from simple communication protocols using encryption and Diffie-Hellman key exchange to OT protocols, a semi-honest two-party GMW protocol defined over a general family of circuits, an n-party coin toss protocol and a hash-check protocol that uses *control flow* non-trivially.

Due to our equational proof style, the proof effort required for our case studies is small. Our example which builds a secure communication channel from an authenticated one using Diffie-Hellman key exchange totals 735 LoC including definitions and all proofs.

References

[1] Bruno Blanchet. Automatic verification of security protocols in the symbolic model: The verifier proverif. In *Foundations of security analysis and design VII*, pages 54–87. Springer, 2013.

[2] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `https://ia.cr/2000/067`.

[3] Ran Canetti, Alley Stoughton, and Mayank Varia. Easyuc: Using easycrypt to mechanize proofs of universally composable security. In *32nd IEEE Computer Security Foundations Symposium*, 2019. `https://eprint.iacr.org/2019/582`.

[4] Andreas Lochbihler, S. Reza Sefidgar, David Basin, and Ueli Maurer. Formalizing constructive cryptography using crypthol. In *32nd IEEE Computer Security Foundations Symposium*, 2019. `http://www.andreas-lochbihler.de/pub/lochbihler2019csf.pdf`.

Formal Verification Challenges on High-Speed Cryptographic Implementations

Miguel Quaresma, Tiago Oliveira

Max Planck Institute for Security and Privacy
{miguel.quaresma,tiago.oliveira}@mpi-sp.org

Jasmin is a framework that enables the development of highly efficient, high-assurance cryptographic implementations [1, 2, 3]. Briefly, the Jasmin framework provides an intuitive programming language to write implementations, a formally verified compiler to compile Jasmin programs into assembly code, and an embedding of the Jasmin programming language into EasyCrypt, to check the correctness and the preservation of security properties of the developed primitives. The compiler also provides several features targeted for the cryptographic domain, specifically safety or constant-time analysis. Jasmin implementations are targeted for a specific CPU architecture, for instance, AMD64. Jasmin implementations utilize particular CPU instructions or instruction set extensions such as the AVX2 to maximize the algorithm's performance. To provide an intuition, AVX2 defines a set of 256-bit registers to perform multiple arithmetic operations at once: for instance, computing the addition between 8 64-bit values using just one instruction.

Kyber [5], one of the NIST PQC candidates, is an interesting case study for some of the challenges that formally verified (high-speed) cryptography presents when using EasyCrypt. The functional correctness proof for the AVX2 implementation follows a 3-hop structure. The extracted EC specification is iteratively modified to obtain simpler (i.e., closer to their scalar counterparts) semantics for the vectorized instructions. This is achieved by replacing the AVX2 instructions with (equivalent) operators that work on arrays of (scalar) values instead of vectors. The semantics of the original implementation are preserved via equivalence proofs between consecutive hops. The last hop is then proven correct w.r.t. the formal Kyber specification and, by the transitive property, equivalent to the extracted EC specification.

This approach simplifies the correctness proof of vectorized implementations by removing the overhead when dealing with AVX2 instructions and allows for the reuse of lemmas/axioms employed by the scalar implementation's correctness proofs. The AVX2 instructions and their scalar counterparts are proved equivalent in a different theory which can be reused by different projects. This approach presents several challenges. On the one hand, a choice has to be made on whether to prioritize instruction semantics over program semantics when specifying scalar operators equivalent to AVX2 instructions. The former simplifies equivalence proofs between instructions and operators. However, it requires additional work when two or more sequential AVX2 instructions work with different size elements for scalar values (e.g., VPADDW followed by VPADDD) while the latter increases the complexity of equivalence proofs for these operators substantially. In addition, despite the approach mentioned above, the complexity of correctness proofs for vector implementations is primarily related to dealing with vectors' semantics and the isomorphisms between these and a list of scalar values.

There are also many exciting verification challenges in the context of scalar implementations, where the code is not vectorized and does not use, for instance, AVX2. An example of such a case is the formal verification of implementations that perform a given arithmetic operation over field elements. In the Curve25519 [4] implementation, field elements can be represented by four 64-bits words (or limbs). During the correctness proof, which also aims to relate a high-level specification with the actual implementation, it is necessary to prove that, for instance, the multiplication that is done using the limbs-representation corresponds to a multiplication that is performed over unbounded integers and that the result remains congruent modulo the prime p being used, in this case, $2^{255} - 19$. It is also necessary to prove that no unexpected overflow occurs. As a challenge, it would be interesting to study how to improve or automate parts of this process, which is, currently, very time-consuming.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: high-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020.

[3] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-assurance cryptography in the spectre era. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 788–805, 2021.

[4] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography - PKC 2006*, pages 207–228, 2006.

[5] Joppe Bos, Leo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*, pages 353–367, 2018.

Formal analysis of LAKE-EDHOC

Elise Klein, Maïwenn Racouchot

January 22, 2022

The Internet Engineering Task Force (IETF) has worked on a solution for Lightweight Authenticated Key Exchange (LAKE), in order to provide a new authenticated protocol [SMP21]. This protocol is called EDHOC, for Ephemeral Diffie-Hellman Over COSE, and its design has been released with a call [VSMW22] for formal analysis and feedback on possible flaws or improvements of EDHOC.

1 Presentation of the LAKE-EDHOC protocol

The EDHOC protocol has been designed to take into account the constraints of the Internet of Things (IoT). Therefore, it tries to reduce the number of messages needed to complete the protocol, the message sizes, as well as the code and memory footprint.

1.1 Protocol design

The protocol relies on a MAC-then-Sign variant for key authentication. The Initiator and the Responder can choose independently to authenticate either using a signature key or a static Diffie-Hellman key, hence providing 4 different modes. Which type of credential is used is decided in the first message of the protocol. In each case, the key exchange is based on ephemeral Diffie-Hellman keys.

The complete protocol consists of three messages and a fourth optional message. Which cryptographic primitives are used is decided through a negotiation. In the first message, the initiator I propose a list of cipher suites that he supports. The responder R triggers an error if he does not support any of the proposition. Otherwise, the protocol continues and R sends the information needed for I to authenticate him. If R is authenticated, I responds with a third message containing her information. A fourth message is possible to explicitly confirm the established Diffie-Hellman key, but it's optional.

1.2 Security goals

The EDHOC protocol aims at ensuring mutual authentication, confidentiality, downgrade protection, identity protection and protection of external data with a security level guaranteed by a minimal size of key. It considers various advanced compromise scenarios giving raise to advanced properties such as forwards secrecy, post compromise security and key compromise impersonation.

2 Our goal

Our work is based on the latest version of the EDHOC protocol. It updates a previous model of EDHOC, in order to prove the security goals enumerated by the IETF task force. Our formal analysis uses the novel SAPIC$^+$ protocol platform, allowing to specify protocols in a dialect of the applied pi calculus which can be automatically translated to and verified by the Tamarin prover [SMCB12] and ProVerif tool [Bla16]. Using both Tamarin and ProVerif allows us to benefit from the strengths of both tools and perform an analysis as comprehensive as possible. We also plan to rely on recent work that relax the usual perfect cryptography assumptions, such as [JCCS19, CJ19].

References

[Bla16] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Found. Trends Priv. Secur.*, 1(1-2):1–135, 2016.

[CJ19] Cas Cremers and Dennis Jackson. Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using diffie-hellman. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 78–93. IEEE, 2019.

[JCCS19] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2165–2180. ACM, 2019.

[SMCB12] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *25th Computer Security Foundations Symposium (CSF 2012)*, pages 78–94. IEEE, June 2012.

[SMP21] Göran Selander, John Preuß Mattsson, and Francesca Palombini. Ephemeral Diffie-Hellman Over COSE (EDHOC). Internet-Draft draft-ietf-lake-edhoc-12, Internet Engineering Task Force, October 2021. Work in Progress.

[VSMW22] Mališa Vučinić, Göran Selander, John Preuß Mattsson, and Thomas Watteyne. Lightweight Authenticated Key Exchange with EDHOC. *IEEE Computer*, April 2022.

Formal analysis of security issues in dynamic virtual networks

Pierre Le Scornet, Université de Rennes 1, IRISA

January 15, 2022

Virtualization is a new development of Cloud technologies. It refers to creating a virtual version of an actual piece of hardware, for example, a real computer with an operating system, and is controlled by a hypervisor running on a host machine. It is often designed to respect some security properties, for example the separation between the virtualized environment and the underlying system. Virtualization is used by Cloud providers to create highly dynamic virtual networks containing various types of virtual machines, and to distribute them to various clients. However, services emerging from this technology become more and more critical to businesses and individuals, and their security weakens as networks grow in size, complexity and dynamicity [6]. For administrators of these complex networks, it is essential to assess risks on their infrastructure to be able to apply appropriate countermeasures.

A key component for such assessments is the concept of *attack graphs*, to represents both static and/or dynamic aspects about the network and the attacker's penetration in it. Attack graphs have already used to model and *verify* network security, i.e. to check whether there exists a sequence of events leading to a security breach. The classical approach to attack graph is to verify the security of *static* networks, i.e. there is a fixed bounded list of machines and the only events are the attacker's actions. We can cite Sheyner et al. [5] and Ou, Boyer, and McQueen [4] *state-based* approach where each node describes a possible state of the network and each arrow is an exploit, Ammann et al. [2] *host-based* approach where each node is a machine and each arrow is an exploit from a source machine to its target machine, and Ammann, Wijesekera, and Kaushik [1] *vulnerability-based* approach where each node is an attribute of the network, for example "is m_1 connected to m_2" or "does the attacker penetrated m", and each arrow is an exploit where the source attribute is necessary for the exploit and the target attribute is "activated" after the exploit. However, these three methods can perform exhaustive security analyses only on static networks, they lack the ability to represent dynamic operations in virtual networks such as virtual machine creations/deletions/migrations. A leap in the direction of dynamic virtual networks was recently made by Mensah [3]: she uses a host-based attack graph model like [2], and she updates dynamically her attack graph when events happen in the network (virtual machine creation/deletion/migration). Her tool is designed to *monitor* the current security risk in the network, so her analysis is not exhaustive. Our contribution is, based on Mensah's model, to verify the security of the virtual network exhaustively. In details:

- We represent the virtual network and its dynamics with a snapshot transition system (STS) where each state represents a snapshot of the virtual network and the attacker's reach in it, and transitions represent the network and attacker's dynamics, such as VM creations/deletions/migrations and attacker's exploits. A snapshot contains VMs, such that each VM holds pieces of information about its hypervisor, its internal configuration (OS, mounted drives, software version…) and the attacker's privilege in it, so snapshots formalize naturally into a finite first-order structure.

- Our goal is to check whether there exists a sequence of events leading to a security breach: it translates naturally as a reachability problem in our STS model. Even if the set of all reachable snapshot is potentially infinite, we prove this reachability problem to be *decidable* under reasonable assumptions, and we discuss the complexity of relevant sub-problems. We especially find the problem to be Pspace-complete for a subclass of reasonable security properties. Finally, we discuss how we can extend our model to a vast majority of the real-life events and how to adapt our proofs to keep our results.

References

[1] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. “Scalable, Graph-Based Network Vulnerability Analysis”. In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. CCS ’02. Association for Computing Machinery, 2002.

[2] Paul Ammann et al. “A host-based approach to network attack chaining analysis”. In: *21st Annual Computer Security Applications Conference (ACSAC’05)*. 2005.

[3] Pernelle Mensah. “Generation and Dynamic Update of Attack Graphs in Cloud Providers Infrastructures”. PhD thesis. CentraleSupélec, 2019.

[4] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. “A Scalable Approach to Attack Graph Generation”. In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 2006.

[5] Oleg Sheyner et al. “Automated generation and analysis of attack graphs”. In: *Proceedings 2002 IEEE Symposium on Security and Privacy*. 2002.

[6] Ivan Studnia et al. “Survey of Security Problems in Cloud Computing Virtual Machines”. In: *Computer and Electronics Security Applications Rendez-vous (C&ESAR 2012). Cloud and security:threat or opportunity*. Nov. 2012. URL: `https://hal.archives-ouvertes.fr/hal-00761206`.

Improving Static Analysis Precision by Minimal Program Refinement

Charles Babu M[1], Sebastien Bardin[1], Matthieu Lemerre[1], and Jean-Yves Marion[2]

[1] Université Paris-Saclay, CEA, List
[2] Université de Lorraine, CNRS, LORIA

Abstract

Imprecision is a very common phenomenon in static analyses that results in false alarms when used for program analysis and program verification. Typically a static analysis approximates the behavior of the code through an abstract domain [2] as checking most program properties is undecidable. Improving precision of a static analysis is an old dream, however, it is complicated. In the last several decades as the size of software grew, the burden of doing manual reasoning became too high, thus marking a trend toward automatic techniques for removing false alarms. Model-checking is one of the very first and among the pioneer on this trend towards which extended the scope of automated techniques through the famous counterexample-guided abstraction refinement (CEGAR) [1] principle. In the static analysis world [5], although there are some works towards automatic techniques for improving precision, either these are not generic and specialized for particular applications, or they rely upon a large part on syntactic heuristics and are prone to path explosion (e.g., trace partitioning [4]).

In this talk we will report on our ongoing efforts to propose, within static analysis, a *syntactic refinement framework* that is *automatic*, *generic*, and *principled.* Through this novel framework, we provide an an algorithmic solution that takes as input a program and outputs a semantically equivalent program on which the static analysis is more precise and complete for a given property, while ensuring minimality of the refinement process – and hopefully avoiding path explosion.

Code obfuscation has attracted attention as an approach to protect a program against reverse engineering. Obfuscating a program with respect to an attacker specified as a static analyzer means making the static analysis of the program imprecise [3]. We believe our refinement framework can help in legitimate obfuscation-related security scenario, such as malware comprehension, by automatically fine-tuning the analysis to the protected code under analysis.

References

1. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: International Conference on Computer Aided Verification. pp. 154–169. Springer (2000)

2. Cousot, P., Cousot, R.: 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. pp. 238–252. ACM (1977)
3. Dalla Preda, M., Giacobazzi, R.: Semantic-based code obfuscation by abstract interpretation. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) Automata, Languages and Programming. pp. 1325–1336. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
4. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: European Symposium on Programming. pp. 5–20. Springer (2005)
5. Rival, X., Yi, K.: Introduction to static analysis: an abstract interpretation perspective. Mit Press (2020)

Learning and Knowledge for Anomaly Detection

Yannick Chevalier

Journées Annuelles du GT Méthodes Formelles pour la Sécurité, 2022

1 Context

Beyond raising an alarm, it is argumented in [4] that a useful Intrusion Detection System (IDS) must have several qualities that are out of the reach of existing systems: The ability to explain an alarm to an operator, the ability to distinguish between a previously unseen normal behaviour and an anomaly, and the ability to distinguish among anomalies those that are accidental and those that are related to an attack. Our first point is that these feature make compliant IDS close to be *Advice Taker (and Giver)* systems in the sense of [3].

Though we are still nowhere close to achieving this kind of intelligent system, previous work on anomaly detection on cars' CAN Bus [1] present a path through which we can get closer to this goal.

2 Logic and the Learning of Logs

We initiated our work on anomaly detection by considering simple characteristics of messages [2]. In spite of the speed of the learning and the monitoring processes, and the fact the we only consider the legitimate messages during the learning phase, we were able in most cases to obtain a better classifier than those based on *Deep Reinforcement Learning.* Our analyser furthermore provided alarms with a *sort of* explainability through a condition that was violated.

We then worked on improving explainations by creating algorithms trying to detect some type of fields within the message, and thus able to better describe the cause of the alarm [2]. We also considered a limited form of *process mining* restricted to `and`-causality, and the detection of a known multi-events protocol, all of which can be encoded in *geometric logic*, a fragment of first-order when all formulas are either positive or implications between positive formulas.

We plan to present a further extension in which the detection algorithms can be any algorithm, in the sense of being a continuous function between Scott Domains. In this extension we define another fragment of first-order logic that is able to both instruct the learner with a partial specification of the system, express the properties learnt during the log analysis, and describe in terms of a set of false literals why an alarm is raised.

References

[1] Chevalier, Y. Data exchange for anomaly detection: The case of the can bus. In *Conference on Artificial Intelligence for Defense*, pages 25–32. MIT Press, 2021.

[2] Chevalier, Y., Rieke, R., Fenzl, F., Chechulin, A., and Kotenko, I. V. Ecu-secure: Characteristic functions for in-vehicle intrusion detection. In Kotenko, I. V., Badica, C., Desnitsky, V., Baz, D. E., and Ivanovic, M., editors, *Intelligent Distributed Computing XIII, 13th International Symposium on Intelligent Distributed Computing, IDC 2019, St. Petersburg, Russia, 7-9 October, 2019*, volume 868 of *Studies in Computational Intelligence*, pages 495–504. Springer, 2019.

[3] McCarthy, J. Programs with common sense. In *Semantic Information Processing*, pages 403–418. MIT Press, 1968.

[4] Sommer, R. and Paxson, V. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE Symposium on Security and Privacy*, pages 305–316, 2010.

Mechanized Proofs of Adversarial Complexity and Application to Universal Composability

Manuel Barbosa
University of Porto & INESC TEC
mbb@fc.up.pt

Gilles Barthe
MPI-SP & IMDEA Software Institute
gbarthe@mpi-sp.org

Benjamin Grégoire
Inria & Université Côte d'Azur
benjamin.gregoire@inria.fr

Adrien Koutsos
Inria
adrien.koutsos@inria.fr

Pierre-Yves Strub
Institut Polytechnique de Paris
pierre-yves@strub.nu

Abstract. We enhance the EasyCrypt with a Hoare logic for reasoning about computational complexity (execution time and oracle calls) of adversarial computations. Our Hoare logic is built on top of the module system used by EasyCrypt for modeling adversaries. We prove that our logic is sound w.r.t. the semantics of EasyCrypt programs.

We showcase (for the first time in EasyCrypt and in other computer-aided cryptographic tools) how our approach can express precise relationships between the probability of adversarial success and their execution time. We exemplify this by revisiting the security proofs of some well-known cryptographic constructions and we present a new formalization of Universal Composability (UC).

Cryptographic Reduction. Cryptographic designs are typically supported by mathematical proofs of security. Unfortunately, these proofs are error-prone and subtle flaws can go unnoticed for many years. Therefore, it is desirable that cryptographic proofs are formally verified using computer-aided tools [8]. EasyCrypt is a proof assistant [3, 4] which has been used to prove security of a diverse set of cryptographic constructions in the computational model of cryptography [1, 2]. In this setting, cryptographic designs and their corresponding security notions are modeled as probabilistic programs. Moreover, security proofs provide an upper bound on the probability that an adversary breaks a cryptographic design, often assuming that the attacker has limited resources that are insufficient to solve a mathematical problem. While EasyCrypt excels at quantifying the probability of adversarial success, it lacks support for keeping track of the complexity of adversarial computations. This limitation means that manual inspection is required to check that the formalized claims refer to probabilistic programs that fall in the correct complexity classes. This work overcomes this limitation and showcases the benefits of reasoning about computational complexity in EasyCrypt, through three broad contributions.

Formal verification of complexity statements. We define a formal system for specifying and proving complexity claims. Our formal system is based on an expressive module system, which enriches the existing EasyCrypt module system with declarations of memory footprints (specifying what is read and written) and cost (specifying which oracles can be called and how often). This richer module system is the basis for modeling the cost of a program as a tuple. The first component of the tuple represents the intrinsic cost of the program, i.e. its cost in a model where oracle and adversary calls are free. The remaining components of the tuple represent the number of calls to oracles and adversaries. This style of modeling is compatible with cryptographic practice and supports reasoning compositionally about (open) programs.

Our formal system takes the form of a Hoare logic for proving complexity claims that upper bound the cost of expressions and commands. Furthermore, an embedding of the formal system into a higher-order logic provides support for reductionist statements relating adversarial advantage and execution cost, for instance:

$$\forall \mathcal{A}. \exists \mathcal{B}.\ \mathsf{adv}_{\mathcal{S}}(\mathcal{A}) \leq \mathsf{adv}_{\mathcal{H}}(\mathcal{B}) + \epsilon \ \land\ \mathsf{cost}(\mathcal{B}) \leq \mathsf{cost}(\mathcal{A}) + \delta$$

where typically ϵ and δ are polynomial expressions in the number of oracle calls. The above statement says that every adversary $\mathcal{A}$ can be turned into an adversary $\mathcal{B}$, with sensibly equivalent resources, such that the advantage of $\mathcal{A}$ against a cryptographic scheme $\mathcal{S}$ is upper bounded by the advantage of $\mathcal{B}$ against a hardness assumption $\mathcal{H}$. Note that the statement is only meaningful because the cost of $\mathcal{B}$ is conditioned on the cost of $\mathcal{A}$, as the advantage of an unbounded adversary is typically 1. The ability to prove and instantiate such $\forall\exists$-statements is essential for capturing compositional reasoning principles.

We show correctness of our logic w.r.t. an interpretation of programs. Our interpretation provides the first complete semantics for the EasyCrypt module system, which was previously lacking.

Implementation in EasyCrypt. We have implemented our formal system in EasyCrypt, providing mechanisms for declaring the cost of operators and for helping users derive the cost of expressions and programs. A key step is to embed our Hoare logic for cost into the ambient higher-order logic—similar to what is done for the other program logics of EasyCrypt. This allows us to combine judgments from different program logics, and it enhances the expressiveness of the approach. Implementation-wise, this extension required to add or rewrite around 8 kLoC. We also exemplify our approach on some classic examples from the EasyCrypt distribution [7] (Bellare and Rogaway, Hashed ElGamal and Cramer-Shoup encryption).

Case study: Universal Composability. Using our enriched implementation of EasyCrypt, we develop a new *fully mechanized* formalization of Universal Composability (UC) [5, 6]. Our mechanization covers the core notions of UC, the classic composition lemmas, transitivity and composability, which respectively state that UC-emulation (as a binary relation between cryptographic systems) is closed under transitivity and arbitrary adversarial contexts. As an example, modular proofs for Diffie-Hellman key exchange and encryption over ideal authenticated channels are composed to construct a UC secure channel.

REFERENCES

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Matthew Campagna, Ernie Cohen, Benjamin Grégoire, Vitor Pereira, Bernardo Portela, Pierre-Yves Strub, and Serdar Tasiran. 2019. A Machine-Checked Proof of Security for AWS Key Management Service. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 63–78. https://doi.org/10.1145/3319535.3354228

[2] José Bacelar Almeida, Cecile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. 2019. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1607–1622. https://doi.org/10.1145/3319535.3363211

[3] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.), Vol. 8604. Springer, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

[4] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings (Lecture Notes in Computer Science)*, Phillip Rogaway (Ed.), Vol. 6841. Springer, 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

[5] Ran Canetti. 2000. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067. (2000). https://eprint.iacr.org/2000/067.

[6] Ran Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 136–145.

[7] The EasyCrypt development team. 2021. Source code of our EasyCrypt. (September 2021). https://github.com/EasyCrypt/easycrypt.

[8] Shai Halevi. 2005. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptol. ePrint Arch.* 2005 (2005), 181. http://eprint.iacr.org/2005/181

Model Checking of Vulnerabilities in Smart Contracts: A Solidity-to-CPN Approach Extended Abstract

Blockchain, which was first introduced as the technology driving Bitcoin, has now outgrown the confines of cryptocurrencies to find its way into a wide range of application areas, including Business Process Management (BPM). Indeed, its intrinsic properties, such as its decentralized structure, capacity to give trust among untrustworthy parties, immutability, and financial transparency, appear to provide the necessary instruments for devising suitable solutions for current BPM issues, particularly for collaborations.

This evolution has been mainly owed to the concept of smart contracts being introduced to Blockchains. A smart contract allows the execution of interdependent transaction sequences while adhering to the rules established in it. On the other hand, a business process may be considered as a set of activities connected by causal relationships with the purpose of attaining a business goal. As a result, smart contracts appear to be excellent candidates for implementing and automating BPs.

Despite significant advancements in Blockchain adoption for BPM, the technology is still in its infancy, and deploying smart contracts to carry out BPs cannot be deemed safe. As a result, proving the correctness of the smart contracts to be deployed on a blockchain is critical for the integrity of the specified business processes.

In our work we propose a formal approach based on the transformation of Solidity smart contracts, with consideration of the BPM context in which they are used, into a Hierarchical Coloured Petri net. We express a set of smart contract vulnerabilities as temporal logic formulae and use the *Helena* model checker to, not only detect such vulnerabilities while discerning their exploitability, but also check other temporal-based contract-specific properties.

Our proposed approach is based on model checking of CPN models and comprises mainly three phases:

1. transforming the smart contracts' Solidity code into CPN submodels corresponding to their functions.
2. transforming the BPM context into a CPN model
3. constructing a CPN model w.r.t an LTL property that can express: *(i)* a vulnerability in the code or *(ii)* a contract-specific property, linking it to a CPN model representing the behavior to be considered, and feeding it the model checker to verify the targeted property.

More precisely, we opt for a hierarchical CPN to represent the considered smart contracts' execution and interaction w.r.t the provided BPM context specification.

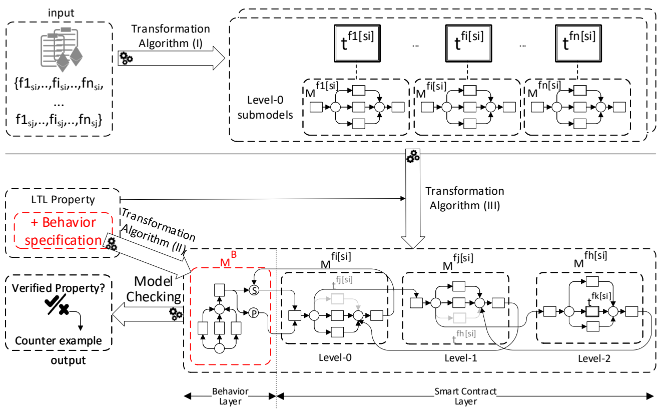

Fig. 1. Overview of the approach

As shown in Figure 1, we represent each function of a smart contract by an *aggregated transition* that encapsulates a submodel corresponding to the internal workflow of the former. In fact, our aim at this first step is to get building blocks for the hierarchical model that will be fed to the model checker. Then, given a context specification (transformed into CPN) and an LTL property to be verified, the final CPN model is built by *(1)* linking the aggregated transition representing the targeted function to the behavioral model and *(2)* building a hierarchy by explicitly representing function calls in the submodel in question (if the checked property requires it).

We have implemented a graphical tool called *Solidity2CPN* that automates the different steps of the proposed approach and makes it accessible to a broader range of users who are unfamiliar with the aspects of formal verification.

Keywords: Blockchain · Business Process Management · Model Checking · Solidity · Smart Contracts · Hierarchical Coloured Petri Nets · Temporal properties.

Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks

Francesco Parolini, Antoine Miné

Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6,
LIP6, 75005 Paris, France

Regular expressions (regexes) are often used to verify that strings in programs match a given pattern. Modern programming languages offer support to regexes in standard libraries, and this encourages programmers to take advantage of them. However, matching engines of languages such as Python, JavaScript, and Java employ algorithms with exponential worst-case time complexity in the length of the string. This is because advanced features such as *backreferences* extend the expressiveness of regular expressions. This comes at the cost of exponential matching in the worst case, even for regexes that do not exploit such features. An attacker can craft a string to force the matcher to exhibit the exponential behaviour to perform a *Regular Expression Denial of Service* (ReDoS) attack, a particular type of *algorithmic complexity attack*.

ReDoS attacks are vastly underestimated Denial of Service (DoS) attacks. In a recent study of regexes usage, in nearly 4,000 Python projects, the authors find that over 42% of them contain regexes [1], while in [2] the authors determine that that up to the 1% of the regexes in open source projects have a superlinear worst-case time behaviour. Staicu et al. [3] showed that hundreds of popular websites are threatened by ReDoS vulnerabilities. Since it is difficult to detect ReDoS vulnerabilities with manual inspection, it is necessary to automate this critical process. However, for now, there is no practical and widely adopted solution to detect ReDoS vulnerabilities.

In this work, we put forward a novel approach to statically detect ReDoS vulnerabilities. We define a *tree semantics* of the matching process, and we leverage it to introduce an analysis that determines whether a regex has a ReDoS vulnerability or not. In particular, the analysis returns an *overapproximation* of the language of words that can cause exponential matching, being effectively *sound* but *not complete*. While, theoretically, it is possible to lose precision, our experiments show that over 1802 regexes our analysis never raises a *false alarm*.

We implemented our algorithm in a tool called `rat`, and we found it to be on average two orders of magnitude faster than most existing detectors, while being proved to be sound and never raising a false alarm in practice. Furthermore, `rat` can extract the language of possibly dangerous words, being more expressive than most other tools.

References

1. Chapman, C., Stolee, K.T.: Exploring regular expression usage and context in Python. In: ISSTA. pp. 282–293. ACM (2016)
2. Davis, J.C., Coghlan, C.A., Servant, F., Lee, D.: The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In: ESEC/SIGSOFT FSE. pp. 246–256. ACM (2018)
3. Staicu, C., Pradel, M.: Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In: USENIX Security Symposium. pp. 361–376. USENIX Association (2018)

Stack smashing analysis by abstract interpretation of binary code

Clément Ballabriga, Julien Forget, Guillaume Person
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
firstname.lastname@univ-lille.fr

I. Context

Binary code analysis has become a popular approach for automated security analysis, as it offers high fidelity reasoning about software behaviour. In comparison, source code analysis requires to make assumptions about the complex compiler behaviour in order to ensure that properties proved for the source code still hold for the compiled binary. Furthermore, in many situations the source code may not be available.

Abstract interpretation enables to compute for each program location an *abstract state* that over-approximates the set of possible (concrete) states at that location. Based on this approximation, we can build tools that automatically *prove the absence* of certain program malfunctions, such as memory corruptions for instance. This contrasts with (and complements) dynamic analyses, that enable to *discover* program security vulnerabilities (e.g. symbolic execution [3], [4]).

II. Goal

In this work, we focus on *stack smashing* vulnerabilities. Stack smashing is a case of stack buffer overflow where an instruction writes to the address containing the return address of a function currently being executed, thus potentially enabling the execution of attacker-controlled code. Our goal is to propose a method that is capable of ensuring the absence of stack smashing vulnerabilities, by fully automated analysis of binary code.

III. Contribution

Our contribution is based on our previous work on relational abstract interpretation of binary code [2]. The analysis proposed in [2] automatically discovers properties relating the contents of the registers and of the memory at each program location. The main originality of this analysis is that it identifies the subset of memory addresses and registers to be represented in the abstract state as the analysis progresses. In comparison, in abstract interpretation of source code the set of variables to analyze is known beforehand as they are declared in the program. Other abstract interpretation techniques for binary code have been proposed, but they are not relational. The use of a relational domain enables to analyse accesses to addresses that are not known statically, and thus to discover a wider range of vulnerabilities than with a non-relational domains (e.g. approaches based on the popular Value Set Analysis [1]).

To analyze stack smashing vulnerabilities, we enrich abstract states with information on the call stack. More precisely, abstract states track the values of the return address of each function call of the stack. Based on this information, we can then check that the return address of a function call is the same at the beginning and at the end of its execution, thus ensuring that no stack smashing can occur.

In our presentation, we will recall the abstract interpretation procedure of [2] and detail its application to stack smashing analysis.

References

[1] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 2732–2733. Springer, 2004.

[2] Clément Ballabriga, Julien Forget, Laure Gonnord, Giuseppe Lipari, and Jordy Ruiz. Static analysis of binary code with memory indirections using polyhedra. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 114–135. Springer, 2019.

[3] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656. IEEE, 2016.

[4] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

Structured Leakage and Applications to Cryptographic Constant-Time and Cost

Gilles Barthe[1], Benjamin Grégoire[2], Vincent Laporte[3], and Swarn Priya[2]

[1]MPI-SP,Germany IMDEA Software Institute, Spain
[2]Université Côte d'Azur, Inria, Sophia Antipolis, France
[3]Université de Lorraine, CNRS, Inria, LORIA

A compiler performs aggressive program optimizations and also respect the input-output behavior of programs. Execution traces define the behavior of programs and compiler correctness is stated as an inclusion between the set of traces of the target program and the set of traces of source programs. However, a correct compiler suffers from three shortcomings in a security context: first, many common security properties are hyperproperties rather than properties; in particular, information flow properties, which cover a broad range of applications are relational properties, i.e. sets of pairs of traces. Second, several security properties of interest, including popular notions of side-channel resistance are modelled by an instrumented semantics that collects (an abstraction of) the adversarially visible physical leakage. Third, the inclusion of instrumented traces fail for most common compiler optimizations, e.g. register allocation and dead code elimination that may add, modify or remove atomic leakages. The main contribution of our work is a novel approach for proving preservation of non-functional properties, and in particular cryptographic constant-time. Cryptographic constant-time is a software countermeasure against cache-based timing attacks, an effective class of side-channel attacks that exploit the latency between cache hits and cache misses to retrieve cryptographic keys and other secrets from program execution. We model leakage using a dedicated data structure that collects atomic leakages. The data structure is closely aligned with the operational semantics of programs. We define a language of leakage transformers that transform leakage of source programs into leakage of target programs. A key benefit of leakage transformers is that they yield an algorithm for computing the leakage of target programs from leakage of source programs. With accurate leakage-transformers at hand, a range of source-level reasoning becomes possible. The structured notion of leakage is used to compute the program's cost, which counts how many times each instruction is executed. The benefit of structured leakage is that it enables computing the cost without looking at the program. Source-level cost analysis with the leakage transformers is used in our work to compute upper-bounds of run-time cost of target programs statically. Our certified cost transformer deduced from the leakage transformers yields an exact cost rather than an upper bound for the generated programs (for all transformations except loop unrolling, where our transformer yields an upper bound). We implement our approach on top of the Jasmin framework. Our contributions in a nutshell:

- the definition of structured leakage and leakage transformers;
- formal proofs of correctness of leakage transformers for all the passes of the Jasmin compiler;
- a proof that the Jasmin compiler preserves CCT;
- a certified algorithm for computing the cost of assembly programs from the cost of Jasmin programs.

All results presented in this work have been formally verified using the Coq Proof Assistant. The complete development is provided in supplementary material [1].

[1]`https://github.com/jasmin-lang/jasmin/tree/constant-time`

Support for Detecting Integer Overflow Vulnerability

Salim Yahia Kissi[1] Yassamine Seladji[1] Rabéa Ameur-Boulifa[2]

[1]LRIT, University of Abou Bekr Belkaid, Tlemcen, Algeria
[2]LTCI, Télécom ParisTech, Institut Polytechnique de Paris, France
{salimyahia.kissi, yassamine.seladji}@univ-tlemcen.dz, Rabea.Ameur-Boulifa@telecom-paris.fr

EXTENDED ABSTRACT.

Organizations and companies develop very complex software today. Errors and flaws can be introduced at different phases of the software development lifecycle and can lead to exploitable vulnerabilities. Furthermore, considering that most systems are exposed to multiple users and environments, such flaws can lead to attacks (or actions) with unpredictable consequences in terms of damage and costs. It is therefore crucial that developers and users know how to detect and prevent them. Despite the substantial knowledge about vulnerabilities nowadays there is still an increasing trend in the number of reported vulnerabilities, that is why software security has become an active research area. this involves various approaches [4, 1] reverse engineering, code review, static and dynamic analysis, fuzzing and debugging.

Increasing scale of software systems requires vulnerabilities scanning tools and supports, that ease their detection and can help coders to avoid them in the development of the code. Furthermore, the use of tools that integrate formal methods might provide evidence for security goals. Towards this end, we have proposed an approach [3] for detecting security bugs which are due to memory safety issues. We were particularly interested in detecting exploits that may be caused by integer overflow. The relevance of our approach lies in the fact that it is based on hardware/software co-analysis. We provided a uniform method for software analysis, considering the specifications of their execution environment (CPUs, compilers, operating systems). The main idea is to build a formula based on the path condition of a given target location in conjunction with the formula (assertions) specifying the environment of its execution, and asking a SMT solver for a satisfying solution, to find out whether the unintended solution is possible. Formally speaking, we use symbolic execution to generate program constraint (PC), and get security constraint (SC) from predefined security requirements. In addition, based on a precise knowledge on the execution context of the analysed program (EC), we propose to solve the statement: $EC \vdash PC \wedge \neg SC$ we seek to find out if there is an assignment of values to program inputs – executed in a certain context – which could satisfy PC but violates SC.

This paper is an extension of our previous work. In this paper, we have significantly extended and implemented the proposed methodology. First, we enrich the set of formulas specifying the effects of memory reference instructions, cross different execution environments to address more platforms. The assertions are derived from various sources, including bad programming practices, compilers configuration settings, operating systems, Common Weakness Enumeration and C standards. Second, we give the technical approach for the vulnerabilities detection: we give details of the tool we are developing for the analysis of C programs. The tool relies on Clang/LLVM compiler infrastructure [2]. This option was motivated by the benefits offered by the infrastructure, including the fact that it supports various target environments such as (x86, x86-64, arm, riscv64, . . .).

Keywords: Formal methods, Security vulnerabilities, Safety bugs.

References

[1] Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Computing Surveys 51*, 3 (2018).

[2] Cadar, C., and Nowack, M. KLEE symbolic execution engine in 2019. *Int. J. Softw. Tools Technol. Transf. 23*, 6 (2021), 867–870.

[3] Kissi., S., Seladji., Y., and Ameur-Boulifa., R. Detection of security vulnerabilities induced by integer errors. In *Proceedings of the 16th International Conference on Software Technologies - ICSOFT,* (2021), INSTICC, SciTePress, pp. 177–184.

[4] Li, H., Kim, T., Bat-Erdene, M., and Lee, H. Software vulnerability detection using backward trace analysis and symbolic execution. *Proceedings - 2013 International Conference on Availability, Reliability and Security, ARES 2013* (2013), 446–454.

Systematic, automated discovery of security protocol attacks that exploit weaknesses in common hash functions

A. Dax, V. Cheval, C. Cremers, L. Hirschi, **C. Jacomme**, S. Kremer

Cryptographic hash functions are a fundamental building block in nearly all cryptographic protocols. They are traditionally required to meet several properties : collision resistance, first- and second-preimage resistance, . . . ; ideally, they also behave "perfectly" and additionally do not suffer from phenomena like length-extension attacks. Modern hash functions like SHA3 (Keccak) are believed to satisfy all these properties and essentially act like so-called random oracles.

In many modern protocol security analyses, both in the computational and symbolic setting, hash functions are assumed to be "perfect" in this sense and act like a random oracle : the modeled hash function meets all desired cryptographic properties, does not suffer from length extension attacks, and each input/output combination is completely independent of any others.

In practice though, real hash functions are unfortunately far from perfect. There are several reasons for this. First, the security of hash functions is often based on a heuristic argument (since we cannot reduce them to a known hard problem) and history has shown that many hash functions that initially seemed secure turned out to be broken some years later. Second, many hash function designs followed the Merkle-Damgård construction, which in its default setup, allows for so-called length extension attacks.

In this work, we formally define a lattice of possible hash weakness encompassing and combining many practical weaknesses of hash functions. We then model the multiple capabilities of the lattice in the symbolic model, and thus how we can use Tamarin or Proverif to verify a protocol for such hash functions modelings. By applying a systematic way to explore all weaknesses and combination of weaknesses with those tools, we were able to identify for multiple real life protocols like IKE, Sigma or SSH both what are the strongest threat models under which the protocol is secure, and the minimal threat models that make the protocol insecure.

Type-directed Program Transformation for Constant-Time Enforcement

Frédéric Besson
Inria
frederic.besson@inria.fr

Thomas Jensen
Inria
thomas.jensen@inria.fr

Gautier Raimondi
Inria
gautier.raimondi@inria.fr

Abstract

Constant-time is a programming discipline which protects cryptographic code against a wide class of timing attacks. This discipline can be formalised as a non-interference property and enforced by an information-flow type-system which prevents branching and memory accesses over secret data. We propose a relaxed information-flow type system which tracks indirect flows but only rejects programs leaking secrets through direct flows. We exploit typing information to guide a program transformation which compiles any well-typed program into a semantically equivalent constant-time program.

1 Introduction

Cryptographic code is notoriously hard to implement. The reason is that it needs to be correct and fast. Yet, functional correctness and speed are not enough. The implementation also needs to be secure with respect to side-channel attacks where attackers attempt to extract cryptographic keys and other confidential information by observing execution time or cache behaviour. *Constant-time programming* is the *de facto* standard to protect implementations against a wide range of timing attacks that exploit micro-architecture side-channels *e.g.*, cache attacks. The constant-time programming discipline imposes further constraints on the code: it is forbidden to make control flow decisions [5] or access memory using secret data as addresses [4].

The present work builds on the insight of Cauligi *et al.* [3] who establish that an information flow type system ensures that a well-typed program can be automatically transformed into a constant-time program. We follow the same methodology but propose a more permissive type-system which allows more programs to be transformed into a constant-time equivalent. The main observation of the paper is that *distinguishing between direct and indirect information flows enables a number of program transformations for constant-time that make it possible to transform program which previous methods would have rejected.*

2 Overview of FaCT's approach

FaCT [3] is a DSL for writing constant-time code. FaCT defines an information flow type system and guarantees that any well-typed program can be transformed into a functionally equivalent but constant-time program.

Predicated code. In order to remove secret control dependencies, FaCT performs a so-called *if-conversion* [1] and generates branchless, predicated code. Consider the code assigning the variable `x` to either `e1` or `e2` depending on a secret `s`

$$\textbf{if } s \textbf{ then } x = e_1 \textbf{ else } x = e_2 \xrightarrow{if-conv.} x = s?e_1 : x; x = !s?e_2 : x$$

After *if-conversion*, we get a branchless code which eliminates the leakage due to the conditional. In terms of information flow, we transformed an indirect flow into a direct flow.

Public Safety. An issue with *if-conversion* is that it is not always a semantics-preserving transformation and can introduce safety issues such as *out-of-bounds* errors.

To solve this issue, the FaCT type system generates verification conditions to ensure that the memory accesses are still valid after transformation *i.e.,* that the expression e in a predicated assignment $x = s?e : e$ is safe to evaluate, independent of the value of s.

3 Indirect Flow Tracking Type System

Compared to a usual information-flow type system, we make a distinction between direct and indirect flows and explicitly record the conditional responsible for indirect flows. Our types extend the usual $\{\mathbf{H}, \mathbf{L}\}$ lattice with an intermediate level $\mathbf{I}(s)$ which represents an indirect flow built from the conditionals identified by the program points in the set s.

$$Type ::= \mathbf{H} \mid \mathbf{L} \mid \mathbf{I}(s) \quad s \in \mathcal{P}(\mathbb{P})$$

Example 1. *Typing Example Consider the following snippet,*

$$(\textbf{if } s@p \textbf{ then } x = e_1 \textbf{ else } x = e_2); y = t[x]$$

the security context $\kappa = \mathbf{L}$ *and the initial typing environment* Γ *such that* $\Gamma \vdash s : \mathbf{H} \quad \Gamma \vdash e : \mathbf{L} \quad e \in \{e_1, e_2, t\}$.

Our type system allows for x *to be typed* $\mathbf{I}(\{p\})$ *after the condition, and therefore the access* $t[x]$ *to be accepted as well, because* $\mathbf{I}(\{p\})$ *is strictly below* $\mathbf{H}$.

4 Type-Directed Program Transformations

As FaCT imposes constraints on its type system, we motivate through example how we enable additional program transformations. All code snippets presented are rejected by FaCT and its implementation.

Delayed if-conversion. Unfortunately, *if-conversion* is not sufficient to remove indirect leakage due to assignments inside the conditional. Consider the following example.

$$\textbf{if } s \textbf{ then } x = e; y = t[x] \textbf{ else skip}$$

Using the classical transformation, we would generate the following insecure code. $x = s?e : x; y = s?t[x] : y$

However, in order to generate a equivalent secure constant-time code, we can perform a *delayed if-conversion* so that the direct flow is generated after the array access. Concretely, our type system accepts the initial code and generates the following secure code.

$$x_t = e; y_t = t[x_t]; x = s?x_t : x; y = s?y_t : y.$$

We enable this transformation by the use of $\mathbf{I}(s)$ to remember that *x is secret due to an indirect flow.*

Out-of-scope Indirect Flows. In the previous case, the leaky memory access is within the scope of the conditional and therefore a delayed *if-conversion* is sufficient to remove the leaky access.

Consider the code snippet of Example 1. The conditional move to x must take place after the memory access $t[x]$ which is outside the scope of the conditional. Our solution is to *associate* the offending memory access with the scope of the conditional responsible for the indirect flow, by using the `next` construction. This way, we can transform it differently depending on if it has a indirect flow just by being in a security context, or by using a insecure variable. We obtain the following code where the memory access is performed after both branches of the conditional, *i.e* always.

$$\textbf{if } s \textbf{ then } x = e_1 \textbf{ else } x = e_2 \textbf{ next } y = t[x]$$

We can then apply the previously shown *delayed if-conversion.*

Safety is Public. In order to prevent *if-conversion* from generating unsafe programs, we assume that the source program is safe and that rendering a program *more safe* does not introduce leakage. As a result, we can instrument array accesses to dynamically ensure that the index are within bounds. The transformation is semantics preserving and is also secure providing the array bounds are public.

5 Ongoing Work

We are currently implementing the presented type-directed transformation within the Jasmin compiler [2]. Proving that the program transformations are correct *i.e.* preserve the observable semantics, should not be problematic and require standard proof techniques. The completeness of the transformation *i.e.* proving that any typable program is successfully compiled into a constant-time program is less standard. We can however notice that the usual C-T type system can be used to verify *a posteriori* the security of the transformation.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of control dependence to data dependence. In J. R. Wright, L. Landweber, A. J. Demers, and T. Teitelbaum, editors, *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, pages 177–189. ACM Press, 1983. doi: 10.1145/567067.567085. URL https://doi.org/10.1145/567067.567085.

[2] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub. Jasmin: High-assurance and high-speed cryptography. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1807–1823. ACM, 2017. doi: 10.1145/3133956.3134078. URL https://doi.org/10.1145/3133956.3134078.

[3] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan. Fact: a DSL for timing-sensitive computation. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 174–189. ACM, 2019. doi: 10.1145/3314221.3314605. URL https://doi.org/10.1145/3314221.3314605.

[4] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, pages 51–65. IEEE Computer Society, 2013. doi: 10.1109/CSF.2013.11. URL https://doi.org/10.1109/CSF.2013.11.

[5] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In D. Won and S. Kim, editors, *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005. doi: 10.1007/11734727_14. URL https://doi.org/10.1007/11734727_14.

Verifying Low-Level, Concurrent Programs with Steel

Aymeric Fromherz[1]

Inria Paris
aymeric.fromherz@inria.fr

Overview. Steel [7, 1] is a framework to develop and prove concurrent programs written in F⋆, a dependently-typed programming language and proof assistant. Inspired by Iris [4, 2, 3], Steel relies on an impredicative concurrent separation logic to reason about concurrent, dependently-typed F⋆ programs. Steel provides several useful features to reason about programs. First, its memory model relies on Partial Commutative Monoids (PCMs), which can encode many programming idioms such as sharing disciplines relying on fractional permissions or references whose contents evolve monotonically according to a user-defined preorder. Additionally, Steel supports dynamically-allocated invariants, which can be shared between threads and allows to reason about lock-free concurrent interactions.

Automation. Specifications in Steel are written using a combination of textbook concurrent separation logic and a restricted form of memory predicates. For instance, Figure 1 provides the signature of a standard swap function. In this specification, the expects and provides annotations correspond respectively to the separation logic pre- and postcondition. The ptr r predicate captures that the reference r is valid, while the separation logic $*$ operator indicates that the two references are separated, that is, they are disjoint in memory. The expects and provides annotations thus capture the shape of memory before and after executing swap. The requires and ensures annotations correspond to *selector predicates*, that is predicates operating on fragments of memory corresponding to the separation logic specification. These predicates allow to specify properties about memory contents such as functional correctness: in this case, that the values of r1 and r2 are swapped after execution.

```
val swap (r1 r2:ref int) : Steel unit
    (expects ptr r1 ∗ ptr r2) (provides λ_ → ptr r1 ∗ ptr r2)
    (requires λ_ → ⊤) (ensures λs _ s' → s'.[r1] = s.[r2] ∧ s'.[r2] = s.[r1])
```

Figure 1: A simple Steel program: swap.

The distinction between separation logic predicates and selector predicates enables us to use different tools to discharge the respective verification conditions. We provide a (partial) decision procedure implemented as an F⋆ tactic to reason about separation logic predicates, while selector predicates are discharged using F⋆'s native support for SMT solvers, thus making verification semi-automated.

Applications. Steel was successfully used to verify a variety of programs [1], including binary self-balancing trees, a monotonic concurrent counter due to Owicki-Gries [6], a racy 2-locks concurrent queue [5], and a library to model message-passing concurrency between two parties on dependently typed channels. Steel aims at being a foundation for verified, low-level systems programming. As such, Steel code currently extracts to verified C code, and we hope to extend our extraction mechanism to also support Rust. In this talk, we propose to give an overview of the Steel framework, as well as current verification projects using it such as the development of a verified memory allocator.

References

[1] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. Steel: Proof-oriented programming in a dependently typed concurrent separation logic. 2021.

[2] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. 2016.

[3] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[4] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. 2015.

[5] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. 1996.

[6] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[7] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. SteelCore: An extensible concurrent separation logic for effectful dependently typed programs. 2020.

Verifying Redundant-Check Based Countermeasures: A Case Study

Thibault Martin
Université Paris-Saclay, CEA, List
Palaiseau, France
thibault.martin@cea.fr

Nikolai Kosmatov
Thales Research and Technology
Palaiseau, France
nikolaikosmatov@gmail.com

Virgile Prevosto
Université Paris-Saclay, CEA, List
Palaiseau, France
virgile.prevosto@cea.fr

Context. Physical attacks of critical embedded systems (via light pulses, laser shots, clock, voltage or electromagnetic glitches, etc.) consist in causing a fault that alters correct execution of software [3, 5]. A frequent goal of such attacks is to bypass some critical checks in the code (such as user authentication, software integrity or software authentication checks) in order to get to a protected point that gives access to sensitive information or physical resources.

To counter such attacks, designers of embedded software use in particular redundancy based countermeasure schemes [3, 4]. In some of these schemes, critical checks (i.e. conditional statements, or tests) in the code are duplicated. In this way, if attackers manage to bypass one check by injecting a fault and flipping the result of the check, the redundant check still prevents them from reaching the protected point. This countermeasure assumes that it is unlikely to inject two faults by physical attacks during the same execution in a coordinated way. It can be generalized to any number $k \geq 1$ of coordinated faults: if an attacker is assumed to be able to introduce k coordinated faults, each critical check should be repeated $k + 1$ times. For simplicity, in the examples of the paper we use $k = 1$.

Examples. A simple C code with a redundant-check countermeasure is illustrated by Fig. 1. Assuming `password` is a user-submitted password and `secret` is the correct password, the duplicated conditional ensures that a bad password will be detected even if one of the conditions is inverted by an attack. Figure 2 shows a more interesting example, with redundant code integrity checks. Such a check is performed by function `check_code_integrity`. As a protection to bit flipping, this function returns a value of the `secbool` type, whose values `sectrue` and `secfalse` have a maximal bit-distance. The second condition is written in a different way, and is erroneous here: the developer should have used a bitwise negation `~chck2` instead of a logical negation `!chck2`. If `chck2` is `secfalse`, its logical negation is in fact 0 so that the test on line 8 is always false. Hence, if an attacker manages to flip the result of only the test on line 6, they will execute the protected line 9 even if code integrity check fails. This example illustrates an incorrect countermeasure, due to a misuse of `secbool` values.

Motivation. Due to their redundant behavior, a correct implementation of countermeasures is difficult to verify, yet crucial to ensure resistance to the considered faults. Various approaches are used to assess the efficiency of countermeasures on a given system. Fault injection based techniques—reproducing potential physical attacks on the target device—allow validation engineers to detect (confirmed) vulnerabilities or get confidence that the system is sufficiently resistant to attacks. Such techniques have the advantage to consider the real-life target system, but remain costly and time-consuming, and cannot guarantee that the system will resist to similar attacks in a slightly different setting (e.g. different signal force, frequency, duration or number of attempts). Another approach consists in searching potential attacks at software level,

```c
if(password != secret) return 1; // Error, bad password
if(password != secret) return 1; // Error, bad password
// Protected: Successfully authenticated
```

Figure 1: Password check with a countermeasure.

```c
1  typedef enum {secfalse = 0x55aa55aa,
2    sectrue = 0xaa55aa55} secbool; // secure true/false values
3  secbool check_code_integrity(); //checks code integrity
4  int main(){
5    secbool chk1=check_code_integrity();
6    if(chk1 != sectrue) return 1; // Error, compromised code
7    secbool chk2=check_code_integrity();
8    if(!chk2 == sectrue) return 1; //incorrect countermeasure
9    // Protected: Successful code integrity check
10 }
```

Figure 2: Integrity check with a countermeasure.

by simulating a chosen set of possible faults in the code and trying to identify potential attacks using test generation [11] or its combination with static analysis [9], or to prove their absence using formal verification [6, 7]. Even if their results are subject to assumptions (about the considered fault model, fault simulation approach, compiler, etc.), software-level approaches provide a useful complement to physical evaluation: they are cheaper, can be fully automatic and can rigorously consider all potential faults with respect to the chosen fault simulation. Such techniques help to find hybrid software/hardware attacks [1].

This study continues previous efforts [6, 7, 9, 11] in this direction. We consider a simple fault model that allows the attacker to invert any subset of at most k checks in the code. "Test inversion" is seen as a very useful mode of fault simulation in a recent joint report by the French certification and evaluation authorities [1, Sec. 16.4].

Contributions. This talk presents a source-code-level formal verification technique of correct implementation of redundant-check based countermeasures. Its purpose is to prove that provoking up to k test inversions in the code should not allow an attacker to reach the protected code. It includes two steps: a dedicated code instrumentation simulating possible faults in critical checks ("test inversions") by mutations; and deductive verification of the resulting code trying to formally prove that the countermeasures effectively prevent attacks. The proposed technique was implemented inside LTest[1] [2], an open-source testing toolset, and relies on the Frama-C[2] verification platform [8]. We evaluated this technique on a real-life case study: the bootloader module of a secure USB storage device called WooKey[3], implemented by the ANSSI and supposed to be resistant to fault injection attacks. We were able to formally prove the correctness of all redundant-check countermeasures in the module except two, and found an error in one of the remaining ones. This error remained undetected despite the fact that this module was rigorously analyzed by 10 evaluation centers[4] as part of a recent evaluation challenge [1]. It confirms the interest of the proposed dedicated approach. This extended abstract presents the work [10] accepted at SAC-SVT 2022.

[1] https://github.com/ltest-dev/LTest
[2] https://frama-c.com
[3] https://wookey-project.github.io/target.html
[4] ITSEFs (Information Technology Security Evaluation Facility), or CESTIs in French

Acknowledgements. This work was partially supported by ANR project SATOCROSS (grant ANR-18-CE25-0015-01). The authors are grateful to ANSSI researchers for presenting WooKey and useful discussions.

REFERENCES

[1] ANSSI and French ITSEFs. Inter-cesti: Methodological and technical feedbacks on hardware devices evaluations. In *SSTIC Symposium*, pages 105–200, 2020.

[2] Sébastien Bardin, Omar Chebaro, Mickaël Delahaye, and Nikolai Kosmatov. An all-in-one toolkit for automated white-box testing. In *Proc. of the 8th International Conference on Tests and Proofs (TAP 2014)*, pages 53–60. Springer, 2014.

[3] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[4] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proc. of the 5th Workshop on Embedded Systems Security (WESS 2010)*. ACM, 2010.

[5] Shivam Bhasin, Paolo Maistri, and Francesco Regazzoni. Malicious wave: A survey on actively tampering using electromagnetic glitch. In *Proc of the 2014 Int. Symp. on Electromagnetic Compatibility*, pages 318–321. IEEE, 2014.

[6] Maria Christofi, Boutheina Chetali, Louis Goubin, and David Vigilant. Formal verification of a CRT-RSA implementation against fault attacks. *J. Cryptogr. Eng.*, 3(3):157–167, 2013.

[7] Karine Heydemann, Jean-François Lalande, and Pascal Berthomé. Formally verified software countermeasures for control-flow integrity of smart card C code. *Comput. Secur.*, 85:202–224, 2019.

[8] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[9] Guilhem Lacombe, David Féliot, Etienne Boespflug, and Marie-Laure Potet. Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. In *Proc. of the 10th International Workshop on Security Proofs for Embedded Systems (PROOFS 2021), colocated with the 2021 Conference on Cryptographic Hardware and Embedded Systems (CHES 2021)*, 2021.

[10] Thibault Martin, Nikolai Kosmatov, and Virgile Prevosto. Verifying redundant-check based countermeasures: A case study. In *Proc. of the 37th Annual ACM/SIGAPP Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT 2022)*. ACM, 2022. To appear. See https://nikolai-kosmatov.eu/publications/martin_kp_sac_2022.pdf.

[11] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *Proc. of the 7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, pages 213–222. IEEE, 2014.

Vérification des mécanismes de sécurité des navigateurs Web

BENJAMIN FARINIER, TU Wien, Autriche

Cette communication présente l'article *WebSpec: Towards Machine-Checked Analysis of Browser Security Mechanisms*, co-écrit avec Lorenzo Veronese, Mauro Tempesta, Marco Squarcina et Matteo Maffei, en cours de relecture à S&P 2022. Une prépublication [1] et les sources [2] du projet sont accessibles en ligne.

Contexte. Les navigateurs Web sont considérés comme faisant partie des logiciels les plus complexes utilisés aujourd'hui, et le nombre de composants de la plateforme Web, c'est-à-dire les fonctionnalités du navigateur et les mécanismes de sécurité, augmente constamment. Ceux-ci sont généralement proposés par les fournisseurs de navigateurs sous la forme d'un brouillon (*W3C Editor's draft*) et discutés au sein de la communauté. Si un consensus suffisant est atteint, le processus de normalisation doit passer par plusieurs niveaux de maturité avant de devenir une recommandation du W3C.

Alors que la mise en œuvre des nouveaux composants de la plateforme Web est soumise à des tests de conformité approfondis [2], leurs spécifications font l'objet d'un examen manuel par des experts pour identifier les problèmes potentiels : il s'agit d'un processus continu et extrêmement complexe qui doit tenir compte de l'interaction avec les API existantes et devrait, en principe, être révisé chaque fois que de nouveaux composants arrivent sur la plateforme Web. Malheureusement, les revues manuelles ont tendance à négliger les failles logiques, ce qui conduit finalement à des vulnérabilités de sécurité critiques. Par exemple, l'en-tête `HttpOnly` a été introduit par Internet Explorer 6 [4] comme moyen de protéger la confidentialité des cookies en ne les exposant pas à des scripts. Huit ans après son lancement, il a été découvert [5] que cette propriété pouvait être trivialement violée par tout script accédant aux en-têtes de réponse d'une requête AJAX via la fonction `getResponseHeader`. Des vulnérabilités de sécurité au niveau des spécifications Web ont également affecté CORS [3], CSP [6] et Trusted Types [1], pour n'en nommer que quelques-uns.

Problème. Cette situation désastreuse découle de plusieurs facteurs concomitants : (1) les composants de la plateforme Web sont spécifiés de manière informelle et, par conséquent, leur analyse, bien que menée par des yeux d'experts, peut facilement ignorer les cas extrêmes ; (2) il n'y a pas de compréhension précise des propriétés de sécurité qui doivent être considérées comme des invariants sur le Web et, par conséquent, être préservées par les mises à jour de la plateforme Web ; (3) les composants de la plateforme Web sont généralement évalués isolément, sans tenir compte de leurs interactions, c'est-à-dire de la nature enchevêtrée de la plateforme Web.

Contributions. Dans ce travail, nous préconisons un changement de paradigme, en soumettant les composants de la plateforme Web et leurs interactions à une analyse de sécurité formelle, par opposition à un examen manuel d'experts. En particulier, nous présentons WebSpec, le premier cadre formel pour l'analyse de la sécurité des mécanismes de sécurité des navigateurs qui prenne en charge la détection automatisée des failles logiques ainsi que les preuves de sécurité vérifiées par machine. WebSpec comprend :

- un modèle de navigateur formel en Coq, qui formalise un ensemble de composants de base de plateforme Web, comprenant à la fois des composants bien établis (cookies, SOP, CORS, etc.) et d'autres récemment introduits (par exemple, CSP3 et Trusted Types) ;
- la définition formelle des dix invariants attendus sur le Web (par exemple, l'intégrité des `__Host`-cookies et le fait qu'une page protégée par CSP ne peut être lue ou modifiée que par les scripts autorisés par la politique) ;
- un compilateur, qui traduit le modèle du navigateur et les invariants en formules SMT afin de permettre la vérification automatique de ces invariants sur le modèle par le solveur Z3. En cas de violation, WebSpec reconstruit la séquence minimale d'actions qui y ont conduit, affichant visuellement l'attaque correspondante.

Nous démontrons l'efficacité de WebSpec par :

- la découverte d'une nouvelle attaque sur les cookies causée par l'interaction avec les anciennes API et d'une nouvelle incohérence entre le CSP et une modification prévue de la norme HTML ;
- la redécouverte de trois failles logiques signalées précédemment dans la plateforme Web actuelle ;
- l'ajustement du modèle pour refléter les états passés de la plateforme Web afin d'identifier cinq attaques précédemment publiées, dans le but de montrer qu'une analyse de sécurité automatisée aurait permis d'éviter ces vulnérabilités ;
- la réalisation de quatre preuves dans le modèle Coq, montrant la justesse de nos modifications proposées pour corriger les vulnérabilités qui affectent actuellement la plateforme Web, y compris une nouvelle technique contre un contournement des Trusted Types.

RÉFÉRENCES

[1] Trusted-types : Restrict to secure contexts. https://github.com/w3c/webappsec-trusted-types/issues/259#issuecomment-630863753.
[2] The web platform tests project. https://web-platform-tests.org/.
[3] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *23rd IEEE Computer Security Foundations Symposium, CSF 2010*, 2010.
[4] O. Community. Httponly cookies. https://owasp.org/www-community/HttpOnly.
[5] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the Incoherencies in Web Browser Access Control Policies. In *31st IEEE Symposium on Security and Privacy, S&P 2010*, 2010.
[6] D. F. Somé, N. Bielova, and T. Rezk. On the Content Security Policy Violations due to the Same-Origin Policy. In *26th International Conference on World Wide Web, WWW 2017*, 2017.

1. https://arxiv.org/abs/2201.01649
2. https://github.com/SecPriv/webspec

List of sponsors

GDR Sécurité Informatique
Le GDR Sécurité Informatique est un outil d'animation de la recherche française créé par l'Institut des sciences de l'information et de leurs interactions (INS2I) du CNRS, et ouvert à toute la communauté. Les thématiques couvertes par le GDR incluent le codage et la cryptographie, les méthodes formelles pour la sécurité, la protection de la vie privée, la sécurité des systèmes, des logiciels et des réseaux, la sécurité des systèmes matériels, la sécurité et les données multimédia.

CEA
Le Commissariat à l'énergie atomique et aux énergies alternatives (CEA) est un organisme public de recherche à caractère scientifique, technique et industriel (EPIC). Acteur majeur de la recherche, du développement et de l'innovation, le CEA intervient dans quatre domaines : la défense et la sécurité, les énergies bas carbone (nucléaire et renouvelables), la recherche technologique pour l'industrie et la recherche fondamentale (sciences de la matière et sciences de la vie). S'appuyant sur une capacité d'expertise reconnue, le CEA participe à la mise en place de projets de collaboration avec de nombreux partenaires académiques et industriels.

Meta
Our mission of giving people the power to build community and bring the world closer together requires constant innovation. That's where research comes in. We believe the most interesting research questions are derived from real-world problems. Our expert teams of scientists and engineers work quickly and collaboratively to build smarter, more meaningful experiences on a global scale by solving the most challenging technology problems, as well as look toward the future.

Author Index