# Adversarial Reachability for Program-level Security Analysis

Soline Ducousso
Univ. Paris-Saclay, CEA, List, Saclay, France
soline.ducousso@cea.fr

In collaboration with Sébastien Bardin and Marie-Laure Potet

***Abstract*—This presentation is based on the paper "Adversarial Reachability for Program-level Security Analysis" from the same authors, accepted for publication at ESOP 2023.**

**Many program analysis tools and techniques have been developed to assess program vulnerability. Yet, they are based on the standard concept of reachability with an attacker able to craft smart *legitimate* input. In practice, attackers can be much more powerful, using for instance micro-architectural exploits or fault injection methods. We introduce *adversarial reachability*, a framework allowing to reason about such *advanced attackers* and assess a program vulnerability to a particular attacker. As equipping the attacker with new capacities significantly increases the state space of the program under analysis, we present a new symbolic exploration algorithm, namely *adversarial symbolic execution*, injecting faults in a *forkless* manner to prevent path explosion, together with optimizations dedicated to reduce the number of injections to consider while keeping the same attacker power. Experiments on representative benchmarks from fault injection show our method significantly reduces the number of adversarial paths to explore, allowing to scale up to 10 faults where prior work timeout for 3 faults. In addition, we analyze the well-tested WooKey's bootloader and demonstrate our analysis' ability to find known attacks and evaluate countermeasures in real-life security scenarios. We were especially able to find a new attack on an incomplete patch.**

***Index Terms*—Program analysis; Attacker model; Fault injection; Symbolic execution**

## I. Introduction

**Context.** Major works have delved into program analysis over the last decades, leveraging techniques such as symbolic execution, static analysis, abstract interpretation, or bounded model checking, to hunt for software vulnerabilities and bugs in programs, or to prove their absence, leading to industrial adoption in some leading companies [1]–[5]. As bugs are an attack entry point, removing them is a first step towards better software security.

**Problem.** Yet, it appears that all these methods consider a rather weak threat model, where the attacker can only craft smart "inputs of death" through legitimate input sources of the program, exploiting corner cases in the code itself. Tools only looking for bugs and software vulnerabilities may deem a program secure while the bar remains quite low for an *advanced attacker*, able for example to take advantage of attack vectors such as (physical) hardware fault injections [6], micro-architectural attacks [7], [8], software-based hardware attacks [9]–[11] like Rowhammer [8], or any combination of vectors [12]. While previously limited to high-security devices and systems such as smart cards and cryptography modules [13], [14], these fault-based attacks now target a wider spectrum of systems, such as bootloaders [15], firmware update modules [16], security enclaves [11], etc. The reasoning behind automated software-implemented fault injection also applies to Man-At-The-End attacks [17] and is similar to the (manual) reasoning performed in control-flow integrity to evaluate countermeasures [18], [19].

**Goal & Challenges.** *Our goal is to devise a technique to automatically and efficiently reason about the impact of an advanced attacker onto a program security properties*, where the standard reachability framework only supports an attacker crafting smart legitimate inputs. The first challenge is to provide a formal framework to study what an advanced attacker can do to attack a program under study. Interestingly, while such frameworks are routinely used in cryptographic protocol verification [20], [21], none has been studied for program-level analysis. The second challenge is to design an efficient algorithm to assess the vulnerability of a program to a given attacker model, while adding capabilities to the attacker naturally give rise to a significant path explosion – especially in the case of multiple fault analysis.

The rare prior work in the field, mostly focused on encompassing physical fault injections for high-security devices, rely mostly on *mutant generation* [22]–[26] or *forking analysis* [12], [27]–[29], yielding scalability issues. Moreover, most of them are limited to a few predefined fault models and do not propose any formalization of the underlying problem.

**Proposal.** We propose *adversarial reachability*, a formalism extending standard reachability to reason about a program execution in the presence of an advanced attacker, and we build a new algorithm based on symbolic techniques, named *adversarial symbolic execution*, to address the adversarial reachability problem from the bug finding point of view (bounded verification). Our algorithm prevents path explosion thanks to a new *forkless* encoding of faults. We show it correct and k-complete with respect to adversarial reachability. To improve the performance further, we design two new optimizations to reduce the number of injected faults: Early Detection of fault Saturation and Injection On Demand.

**Contributions.** Our main contributions are the following:

- We formalize the adversarial reachability problem, extending standard reachability with new faulted transitions to take into account an advanced attacker, together with the associated correctness and completeness definitions;
- We describe a new symbolic exploration method, adversarial symbolic execution, to answer adversarial reachability. It features a novel forkless fault encoding wrapping arithmetically the expression to be faulted to prevent path explosion , but it increases query complexity and causes scaling issues. To mitigate it, we designed two optimization strategies to reduce fault injection: the first one stopping as early as possible, Early Detection of fault Saturation (EDS) and the other starting the injection as late as possible, Injection On Demand (IOD). We establish their correctness and completeness;
- We propose an implementation of our techniques for binary-level analysis, on top of the BINSEC framework [30]. We systematically evaluate its performances against prior work, using a standard SWiFI benchmark from physical fault attacks and smart cards. Experiments (Figure 1) show a very significant performance gain against prior approaches, for example up to x10 and x215 times on average for 1 and 2 faults respectively – with a similar reduction in the number of adversarial paths. At most, we are x224 times faster for 1 fault and x6000 for 2. Moreover, our approach scales up to 10 faults whereas the state-of-the-art starts to timeout for 3 faults.
- We finally perform a security analysis of the WooKey bootloader [1], a very well tested real-life security focused program. We were able to find various known attacks [12], [33] and evaluate the adequacy of some of the countermeasures. Especially, we found an attack not mentioned before on a recently proposed patch [12], and proposed a patch to the developers.

This work is a first step in designing efficient program analysis techniques able to take into account advanced attackers. The approach is generic enough to accommodate many common fault models, including the bit flip from RowHammer, test inversion or arbitrary data modification; still, instruction skips or modifications are currently out of reach. Also, while we investigate the bug finding side of the problem (underapproximation), the verification side (overapproximation) is interesting as well. These are exciting directions for future research.

## II. Software-implemented faults injection (SWiFI)

### A. Related work

SWiFI tools [12], [22]–[29], [33] have been developed in the community of high-secure systems to ease time-consuming hardware fault injection campaigns. SWiFI evaluates a program with the transformations induced by the effects of hardware faults, in order to find interesting attack paths. We distinguish two main SWiFI techniques. First, the *Mutant generation* approach [22]–[26] consists in creating slightly modified versions of the program (named mutants), each of them embedding a different faulty instruction. Each mutant is analyzed on its own, typically with symbolic execution. Second, the *forking approach* [12], [27]–[29] consists in instrumenting the analysis (or the code, via instrumentation) to add all possible faults as forking points (branches) controlled by boolean values indicating whether a particular fault will be taken or not, plus constraints on the maximal number of faults allowed. A standard program analysis technique is then launched – typically symbolic execution or bounded model checking.

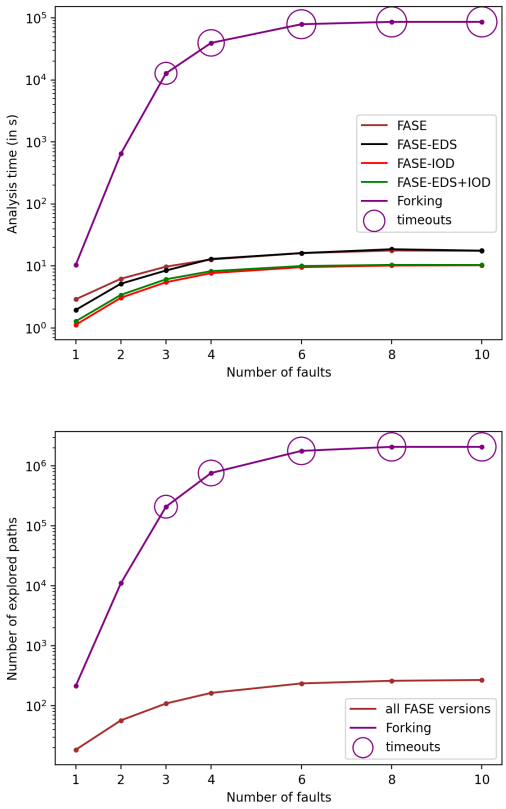


Fig. 1: Experimental results comparing our technique FASE and its variants with another approach, Forking on three metrics for arbitrary data faults.

**Scalability issues** These two approaches yield an explosion of the whole search space w.r.t. the number of fault injection points in the program: the mutant approach leads to consider up to $C_k^n$ ($k$ among $n$)[2] mutants for a program with $n$ possible fault locations and $k$ faults, while the forking approach yields up to $C_k^n$ paths to analyzed for a single original program path. Only few SWiFI tools can handle multiple faults [12], [27], [33], [34] – still with scalability issues.

[1] WooKey [31], [32] is a secure USB mass storage device developed by the French National Security Agency, and has recently served as a recent challenge among French security evaluators.

[2] Remind that $C_k^n = \binom{k}{n} = \frac{n!}{k!(n-k)!}$

### B. Forkless Adversarial Symbolic Execution (FASE)

We design FASE with the following guiding principles:

- Prevent path explosion as much as possible with a new forkless fault encoding wrapping arithmetically an expression. It is easily adapted to other data fault models and test inversion.
- FASE should be correct and k-complete with respect to adversarial reachability.
- Reduce as much as possible the complexity of the created formulas by avoiding the undue introduction of extra-faults along a path. Our two optimizations only remove impossible attacker behavior and thus remain correct and k-complete for adversarial reachability.

## III. Conclusion

We formalize the concept of adversarial reachability, extending standard reachability to include an advanced attacker in program analysis, and we propose a dedicated symbolic algorithm, integrating a novel forkless encoding of faults together with dedicated optimizations. Our technique is shown to significantly reduce the number of paths to explore, and scales up to 10 faults on a standard SWiFI benchmark, where prior forking attempts timeout for 3 faults. Also, we show our method scale to realistic size examples, such as the WooKey project where we have been able to find known fault attacks and to even find a vulnerability not mentioned before in a recently proposed patch [12].

## References


[1] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *ICSE*, 2013.

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

[3] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "Slam and static driver verifier: Technology transfer of formal methods inside microsoft," in *iFM*, 2004.

[4] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Form. Asp. Comput.*, 2015.

[5] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn, "Finding real bugs in big programs with incorrectness logic," no. OOPSLA, 2022.

[6] D. Karaklajić, J.-M. Schmidt, and I. Verbauwhede, "Hardware designer's guide to fault attacks," *IEEE VLSI Systems*, 2013.

[7] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *SP*, 2019.

[8] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *TCAD*, 2019.

[9] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *USENIX Security*, 2017.

[10] J. Gravellier, J.-M. Dutertre, Y. Teglia, and P. L. Moundi, "Faultline: Software-based fault injection on memory transfers," in *HOST*, 2021.

[11] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *SP*, 2020.

[12] G. Lacombe, D. Feliot, E. Boespflug, and M.-L. Potet, "Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities," in *PROOFS WORKSHOP*, 2021.

[13] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *EUROCRYPT*, 1997.

[14] G. Barthe, F. Dupressoir, P.-A. Fouque, B. Grégoire, and J.-C. Zapalowicz, "Synthesis of fault attacks on cryptographic implementations," in *CCS*, 2014.

[15] J. Van den Herrewegen, D. Oswald, F. D. Garcia, and Q. Temeiza, "Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis," *TCHES*, 2021.

[16] C. Bozzato, R. Focardi, and F. Palmarini, "Shaping the glitch: optimizing voltage fault injection attacks," *TCHES*, 2019.

[17] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat, and M. K. Khan, "Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions," *J. Netw. Comput. Appl.*, 2015.

[18] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *TISSEC*, 2009.

[19] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *CSUR*, 2017.

[20] I. Cervesato, "The Dolev-Yao intruder is the most powerful attacker," in *LICS*, 2001.

[21] G. Bana and H. Comon-Lundh, "A computationally complete symbolic attacker for equivalence properties," in *CCS*, 2014.

[22] M. Christofi, B. Chetali, and L. Goubin, "Formal verification of an implementation of crt-rsa vigilant's algorithm," in *PROOFS workshop: pre-proceedings*, 2013.

[23] P. Rauzy and S. Guilley, "A formal proof of countermeasures against fault injection attacks on crt-rsa," *JCEN*, 2014.

[24] T. Given-Wilson, N. Jafri, J.-L. Lanet, and A. Legay, "An automated formal process for detecting fault injection vulnerabilities in binaries and case study on present," in *Trustcom/BigDataSE/ICESS*, 2017.

[25] S. Carré, M. Desjardins, A. Facon, and S. Guilley, "Openssl bellcore's protection helps fault attack," in *DSD*, 2018.

[26] T. Given-Wilson, N. Jafri, and A. Legay, "Combined software and hardware fault injection vulnerability detection," *Innov. Syst. Softw. Eng.*, 2020.

[27] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections," in *ICST*, 2014.

[28] M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant, "Idea: embedded fault injection simulator on smartcard," in *ESSoS*, 2014.

[29] J.-B. Bréjon, K. Heydemann, E. Encrenaz, Q. Meunier, and S.-T. Vu, "Fault attack vulnerability assessment of binary code," in *CS2*, 2019.

[30] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion, "Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis," in *SANER*, 2016.

[31] R. Benadjila, M. Renard, P. Trebuchet, P. Thierry, A. Michelizza, and J. Lefaure, "Wookey: Usb devices strike back," *SSTIC*, 2018.

[32] https://github.com/wookey-project, accessed july 2021.

[33] T. Martin, N. Kosmatov, and V. Prevosto, "Verifying redundant-check based countermeasures: a case study," in *SAC*, 2022.

[34] S. Winter, M. Tretter, B. Sattler, and N. Suri, "simfi: From single to simultaneous software fault injections," in *DSN*, 2013.