Presentation: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level with BINSEC/REL

Lesly-Ann Daniel[1], Sébastien Bardin[1], and Tamara Rezk[2]

[1]CEA List, Université Paris-Saclay
[2]INRIA Sophia-Antipolis, INDES Project
lesly-ann.daniel@cea.fr, sebastien.bardin@cea.fr, tamara.rezk@inria.fr

Published work at IEEE Symposium on Security and Privacy 2020.
`https://arxiv.org/pdf/1912.08788.pdf`

Connection with Klee: This works builds on the idea proposed in Klee and Shadow Symbolic Execution *Shadow Symbolic Execution* [1] of executing two versions of a program in the same symbolic execution instance while maximizing sharing between them. We show that this sharing fails on binary code and design new optimizations for binary analysis. This can in turn be relevant to Klee in the context of information flow analysis.

Context. Timing attacks exploit correlations between secret data and the execution time in order to recover information about the secrets manipulated by a program (e.g. sensitive data, secret keys). These correlations can happen when the *control-flow* depends on the secrets, but also through secret-dependent *memory accesses* via cache attacks. *Constant-time* (CT) is a software-based countermeasure to timing attacks which is already widely used to secure cryptographic implementations (e.g. BearSSL, NaCL, HACL*, etc). It requires the control flow and the memory accesses of the program to be independent from the secret input. Formally, it states that *for each pairs of execution traces t and t' with the same public input, t and t' must have the same control flow and the same memory accesses*, regardless of the value of the secrets.

Problem. Constant-time is generally not preserved by compilers [2]. For example, reasoning about constant-time requires to know whether the code `c=(x<y)-1` will be compiled to branchless code, but this depends on the compiler version and optimization [2]. Several CT-analysis tools have been proposed to analyze source code [3], or LLVM code [4], but leave the gap opened for violations introduced by compilers [2]. Binary-level tools for constant-time based on dynamic analysis [5] can find bugs but are incomplete; while static approaches [6] can prove that a program is secure but cannot report precise bugs. *Our goal is to fill this gap and design an efficient tool for bug-finding and bounded-verification of constant-time at binary-level.*

Challenge of constant-time analysis is twofold. First, standard verification methods do not directly apply because constant-time is not a regular safety property but a 2-hypersafety property [7] (i.e. relating two execution traces). Second, it is notoriously difficult to adapt formal methods to binary-level because of the lack of structure information (data and control) and the explicit representation of the memory as a large array of byte.

Symbolic execution for 2-hypersafety. *Symbolic execution* (SE) [8], [9] is widely used for bug-finding and can also perform bounded-verification. However its direct adaptation to 2-hypersafety via self-composition does not scale. An interesting idea is to represent *two execution traces in the same symbolic execution instance* while maximizing sharing between them. This idea has first been introduced as *ShadowSE* [1] in the context of back-to-back testing, then adapted later in the context of 2-hypersafety as *relational symbolic execution* [10].

Relational symbolic execution (RelSE) [10] models two execution traces in the same symbolic execution instance by mapping variables to either *pairs of symbolic expressions* $\langle \varphi \,|\, \varphi' \rangle$ when they *may depend* on the secret; or to *simple expressions* $\langle \varphi \rangle$ when they *do not depend* on the secret. To analyze constant-time, we send a query to an smt-solver at each conditional statement and each memory access which evaluates to a pair of expression, to ensures that it does not depend on the secret. On the other hand in the case of a simple expression which, by definition, does not depend on the secret, the query can be spared. RelSE enables sharing of public data – which is equal in both executions – and reduces the number of solver calls by tracking secret-dependencies.

Unfortunately, we show that direct adaptation of RelSE does not scale for binary analysis. In binary analysis, the memory is explicitly represented as a symbolic array of bytes and is duplicated at the beginning of the execution, hence hindering sharing and secret-tracking.

Proposal. We tackle the problem of designing an efficient symbolic verification tool for constant-time at binary-level that leverages the full power of symbolic execution without sacrificing correct bug-finding nor bounded-verification. Our technique builds on *relational symbolic execution* which we complement with *three dedicated optimizations* for binary-level and constant-time analysis. In particular, one of our optimizations named *FlyRow* is dedicated to binary-level RelSE. *FlyRow* builds on *read-over-write* [11] and resolves most load operations on-the-fly, which avoids to resort to the duplicated memory and enables precise secret-dependency tracking.

We present BINSEC/REL, the first efficient binary-level automatic tool for bug-finding and bounded-verification of constant-time at binary-level. It is compiler agnostic, targets x86 and ARM architectures and does not require source code. BINSEC/REL can analyze about 23 million instructions in 98 min (i.e. 3860 instructions per second) while being still correct and complete for constant-time.

Contributions. Our contributions are the following:

- We design dedicated optimizations for information flow analysis at binary-level. Moreover, we formally prove that our analysis is correct for bug-finding and bounded-verification of constant-time.
- We propose a verification tool named BINSEC/REL for constant-time analysis and perform an experimental evaluation against standard approaches based on self-composition and RelSE on 338 cryptographic binaries. We show that BINSEC/REL is 700 times faster that RelSE and can achieve bounded-verification on large programs where standard approaches timeout.
- We perform bounded-verification on 296 constant-time cryptographic binaries previously verified at a higher level (e.g. BearSSL, NaCL, HACL*), and replay 42 known bugs.
- Finally, we extend a manual study on constant-time preservation by compilers [2] : (1) we automatically analyze the code that was manually checked in [2], (2) we add new implementations, (3) we add the `gcc` compiler and a more recent version of `clang`, (4) we add ARM binaries. Interestingly, we discovered that `gcc -O0` and backend passes of `clang` can introduce violations out of reach of LLVM verification tools like ct-verif [4]. This shows the importance of performing constant-time analysis at binary level.

References

[1] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a doubt: Testing for divergences between software versions", in *ICSE*, 2016.

[2] L. Simon, D. Chisnall, and R. J. Anderson, "What you get is what you C: controlling side effects in mainstream C compilers", in *EuroS&P*, 2018.

[3] S. Blazy, D. Pichardie, and A. Trieu, "Verifying constant-time implementations by abstract interpretation", in *ESORICS*, 2017.

[4] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time Implementations.", in *USENIX*, 2016.

[5] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software", in *USENIX*, 2017.

[6] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels", *ACM Transactions on Information and System Security*, 2015.

[7] M. R. Clarkson and F. B. Schneider, "Hyperproperties", *Journal of Computer Security*, 2010.

[8] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing", *Communications of the ACM*, 2012.

[9] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later", *Communications of the ACM*, 2013.

[10] G. P. Farina, S. Chong, and M. Gaboardi, "Relational symbolic execution", in *PPDP*, 2019.

[11] B. Farinier, R. David, S. Bardin, and M. Lemerre, "Arrays made simpler: An efficient, scalable and thorough preprocessing", in *LPAR*, 2018.