llvm2CryptoLine: Verifying Arithmetic in Cryptographic C Programs

Ruiling Chen
Shenzhen University
Shenzhen, China
chenchajun5555@gmail.com

Jiaxiang Liu*
Shenzhen University
Shenzhen, China
jiaxiang0924@gmail.com

Xiaomu Shi
Institute of Software, CAS
Beijing, China
xshi0811@gmail.com

Ming-Hsien Tsai
National Institute of Cyber Security
Taipei, Taiwan
mhtsai208@gmail.com

Bow-Yaw Wang
Academia Sinica
Taipei, Taiwan
bywang@iis.sinica.edu.tw

Bo-Yin Yang
Academia Sinica
Taipei, Taiwan
byyang@iis.sinica.edu.tw

ABSTRACT

Correct implementations of cryptographic primitives are essential for modern security. These implementations often contain arithmetic operations involving non-linear computations that are infamously hard to verify. We present llvm2CryptoLine, an automated formal verification tool for arithmetic operations in cryptographic C programs. llvm2CryptoLine successfully verifies 51 arithmetic C programs from industrial cryptographic libraries OpenSSL, wolfSSL and NaCl. Most of the programs are verified fully automatically and efficiently. A screencast that showcases llvm2CryptoLine can be found at https://youtu.be/QXuSmja45VA. Source code is available at https://github.com/fmlab-iis/llvm2cryptoline.

CCS CONCEPTS

• **Software and its engineering → Formal software verification.**

KEYWORDS

formal verification, cryptographic programs, functional correctness

ACM Reference Format:
Ruiling Chen, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2023. llvm2CryptoLine: Verifying Arithmetic in Cryptographic C Programs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3611643.3613096

1 INTRODUCTION

Cryptographic primitives are building blocks of modern computer security. They are usually confronted with the most severe adversarial environments compared to other programs, making their correctness notably important. Modern cryptography relies on complicated mathematics, and furthermore, should be implemented on real-world hardware architectures. Take the curve Curve25519 [1] used in Elliptic Curve Cryptography (ECC) [12, 15] as an example. It is defined over the finite field $\mathbb{Z}_{2^{255}-19}$. Hence even computing the sum of two field elements consists of two operations, that is, addition and modulo. Thousands of field arithmetic operations are required during key generation, for instance, in OpenSSH. Note that a field element in $\mathbb{Z}_{2^{255}-19}$ is representable by at least 255 bits. However, no computers can directly perform 255-bit arithmetic operations. Typical implementations in 64-bit architectures employ four 64-bit or five 51-bit numbers to represent a field element. Arithmetic over the finite field is implemented on such representations and highly optimized for efficiency. The complexity from both mathematics and implementation makes it difficult to conclude the functional correctness of these cryptographic programs. Bugs are found by formal techniques even for these basic, exhaustively tested field operations [14, 21]. Formal guarantees are thus necessary.

Cryptographic primitives are generally implemented in low-level languages like C. Numerous promising tools are available for formally verifying C programs (e.g., [2, 3, 6, 8, 9, 13, 19, 20]), many of which are built on abstraction and/or SMT solving. Our previous work has shown that these general-purpose C verification tools are not very suitable for cryptographic programs, since *bit-precise* analysis of *non-linear* computations is required when verifying the arithmetic in cryptographic programs [14]. Dedicated verification techniques are demanded. CryptoLine is a domain-specific language equipped with a tool for modeling and verifying cryptographic assembly programs [7, 17]. We proposed in [14] to verify cryptographic C programs via translating their intermediate representations (IRs) extracted from the Clang compiler into CryptoLine programs, and then verifying them using the CryptoLine tool. We have empirically identified a core subset llvmCrypto of LLVM IR for cryptographic programs and modeled programs representable by llvmCrypto with the CryptoLine language.

In this paper, we present a verification tool llvm2CryptoLine for cryptographic C programs, which extends and realizes the approach proposed in [14]. Specifically, the approach in [14] models IR programs with the *untyped* CryptoLine language [17], where all variables and constants are unsigned and of the same bit-width. In this work, we instead model IR programs with the *typed* CryptoLine language [7], where both signed and unsigned representations as well as different bit-widths are available. It allows to verify cryptographic C programs with not only unsigned types but signed

*Corresponding author

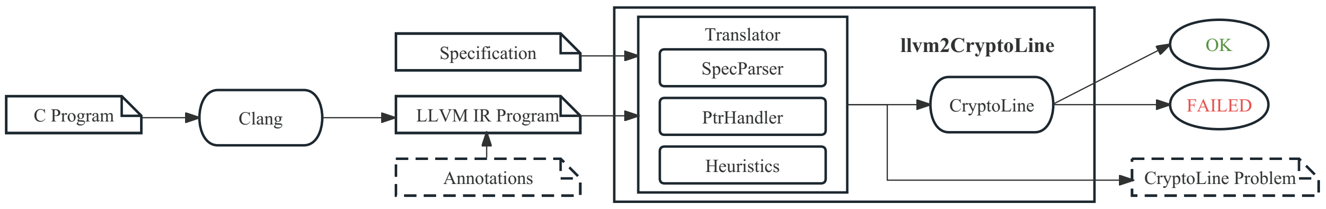

Figure 1: Architecture and Workflow of llvm2CryptoLine

types, hence improving the applicability of our approach. Secondly, we enlarge the subset llvmCrypto to support more data types (e.g., arrays and structures) and more instructions (e.g., signed extension and type casting), further broadening the applicability. On the other hand, the approach in [14] requires users to write specifications in the generated CryptoLine programs, that is, to specify what properties should be satisfied by the CryptoLine program, thus the IR program and the target C program. This compels users to learn the CryptoLine language and to understand the translation from llvmCrypto to CryptoLine. In this work, we design and implement a specification language that enables users to write specifications directly at the LLVM IR level. It offers better usability and is more friendly for beginners. Last but not least, manual intervention and annotations in generated CryptoLine programs are sometimes needed due to the limitations of the CryptoLine tool. We develop heuristics in llvm2CryptoLine to reduce human efforts, hence further improving usability.

For evaluation, we have successfully employed llvm2CryptoLine to verify 51 C implementations for arithmetic operations. They come from cryptographic primitives in OpenSSL [18], wolfSSL [5] and NaCl [4]. 13 of them are not supported by the approach in [14]. llvm2CryptoLine verifies 29 out of 51 functions fully automatically, and verifies most of them within merely 1 minute.

We summarize our contributions as follows:

- We present llvm2CryptoLine, an automated cryptographic C verification tool that supports both signed and unsigned types, and allows specifications at the LLVM IR level.
- We verify with llvm2CryptoLine 51 arithmetic C programs from industrial libraries OpenSSL, wolfSSL and NaCl.

2 THE LLVM2CRYPTOLINE TOOL

The architecture and workflow of llvm2CryptoLine are depicted in Figure 1. llvm2CryptoLine accepts as inputs a specification and an LLVM IR program, which is extracted from the target C program compiled by the Clang compiler. It then *automatically* outputs whether the expected specification is satisfied by the given IR program ("OK"), thus by the target C program, or unknown ("FAILED"). The core of llvm2CryptoLine is the module `Translator`. It models LLVM IR instructions with typed CryptoLine instructions, translating the input IR program and specification to a CryptoLine program with specification that can be handled by the CryptoLine tool. llvm2CryptoLine then invokes CryptoLine to solve the generated CryptoLine problem. It answers "OK" if CryptoLine successfully verifies the problem. Otherwise, "FAILED" is returned indicating that whether the specification holds or not is unknown. In this case, human intervention is needed. Users can add extra hints about the target program by augmenting the IR program with annotations. Or experienced users can ask llvm2CryptoLine to output the generated CryptoLine problem and inspect it directly.

The module `Translator` moreover includes three important submodules `SpecParser`, `PtrHandler` and `Heuristics`. `SpecParser` parses the input specification written in our specification language (Section 2.2). `PtrHandler` analyzes pointers and memory accesses in the IR program, and models them using the CryptoLine language that does not support these features (Section 2.3). `Heuristics` applies heuristic rules to improve the automation of llvm2CryptoLine (Section 2.4). That is, it reduces the burden on users of writing annotations or investigating the CryptoLine problem.

2.1 llvmCrypto

LLVM IR is so general that programs in many languages can be represented through dedicated compiler frontends. We have identified in [14] a core subset called llvmCrypto for cryptographic C programs. In this work, we further enrich llvmCrypto to support more cryptographic programs. The llvmCrypto implemented in llvm2CryptoLine supports data types including integers, vectors, pointers, arrays and structures. The pointer type is restricted to only point to integers, vectors, arrays, structures, as well as the pointers to these types (i.e. double pointers). llvmCrypto allows arithmetic operations addition (`add`), subtraction (`sub`) and multiplication (`mul`); as well as bitwise operations left-shift (`shl`), logical right-shift (`lshr`), arithmetic right-shift (`ashr`), bitwise AND (`and`), OR (`or`) and exclusive-OR (`xor`). The vector versions of these operations are also defined. It moreover provides instructions `insertelement` and `extractelement` to manipulate vectors. For memory- and pointer-related operations, the instructions `getelementptr`, `alloca`, `load`, and `store` are supported, where `getelementptr` performs pointer arithmetic. Lastly, to accommodate C programs with signed types, conversion operations like truncation (`trunc`), zero extension (`zext`), sign extension (`sext`) and type casting (`bitcast`) are included. Note that there are no control-flow instructions such as branching in llvmCrypto, because such instructions are avoided in typical cryptographic programs for side-channel attack prevention [16].

One may notice that llvm2CryptoLine in fact verifies the correctness of IR programs instead of C programs. The benefits of this design decision are threefold. Firstly, IR code is simpler and more structured than C code, making the analysis more manageable. Secondly, IR code is closer to the actual executable than C code. Verifying IR code instead of C code excludes the compiler frontend from the *trusted computing base*, providing stronger guarantees on the correctness of cryptographic programs. Thirdly, LLVM IR is language-independent. Working at the IR level grants our approach the extendability to verify cryptographic programs written in other languages that can be compiled to LLVM IR (e.g., Rust and Swift).

$$
\begin{aligned}
&\mathit{Num} ::= \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \qquad \mathit{Width} ::= 1 \mid 2 \mid \cdots \\
&\mathit{Var} ::= x \mid y \mid z \mid x' \mid y' \mid z' \mid \cdots \qquad \mathit{Const} ::= \mathit{Num}@\mathit{Width} \\
&\mathit{Atom} ::= \mathit{Var} \mid \mathit{Const} \\
&\mathit{Expr} ::= \mathit{Atom} \mid \mathit{Expr} + \mathit{Expr} \mid \mathit{Expr} - \mathit{Expr} \mid \mathit{Expr} * \mathit{Expr} \\
&\mathit{APred} ::= \mathtt{true} \mid \mathit{Expr} = \mathit{Expr} \mid \mathit{Expr} \equiv \mathit{Expr} \bmod \mathit{Expr} \mid \mathit{APred} \land \mathit{APred} \\
&\mathit{RPred} ::= \mathtt{true} \mid \mathit{Expr} = \mathit{Expr} \mid \mathit{Expr} <_u \mathit{Expr} \mid \mathit{Expr} \leq_u \mathit{Expr} \\
&\qquad \mid \mathit{Expr} <_s \mathit{Expr} \mid \mathit{Expr} \leq_s \mathit{Expr} \mid \mathit{RPred} \land \mathit{RPred} \\
&\mathit{Cond} ::= \mathit{APred} \mathbin{\&\&} \mathit{RPred} \qquad \mathit{Spec} ::= \{\ \mathit{Cond}\ \}\ \{\ \mathit{Cond}\ \}
\end{aligned}
$$

Figure 2: llvm2CryptoLine Specification Language

2.2 Specification Language

We design and implement a specification language to describe functional correctness properties that target programs should satisfy. The syntax is shown in Figure 2. Atoms (*Atom*) are either variables (*Var*) or constants (*Const*) with bit-widths (*Width*). An expression (*Expr*) is an atom, or the sum, difference, product of expressions. Two kinds of predicates are defined. An algebraic predicate (*APred*) is a conjunction of equations or modulo equations. It characterizes algebraic properties of the target program. On the other hand, a range predicate (*RPred*) is a conjunction of signed or unsigned comparisons, describing range properties. A condition (*Cond*) is divided into an algebraic part and a range part. Finally, a specification (*Spec*) consists of a *precondition* and a *postcondition*. A specification is satisfied by a program, if the program always ends in a state conforming to the postcondition whenever it starts from a state meeting the precondition [10]. llvm2CryptoLine automatically checks whether the input IR program satisfies the input specification.

It is stipulated that the variables specified in the precondition must be the parameters of the target function. Take the following (simplified) IR function from Curve25519 in OpenSSL as an example:

```
void @fe51_add(i64* %h, i64* %f, i64* %g)
```

The variables in the precondition must be `h`, `f`, or `g`. Typical implementations of arithmetic operations in cryptography represent mathematical elements as arrays. `h`, `f` and `g` here are arrays of size 5 representing field elements in $\mathbb{Z}_{2^{255}-19}$. The resulting field element of function `fe51_add` is stored in `h`. Our specification language allows C-like notation to access elements in arrays. `h[0]` denotes the 0th element in `h`. For postconditions, variables must be the function's parameters, with or without prime symbols (`'`). The prime symbol is reserved for the *final* (or *output*) value of a variable after executing the function. So `h'[0]` denotes the value of element `h[0]` when `fe51_add` returns. Correspondingly, a variable without the prime stands for its *initial* value before executing the function.

Additionally, syntactic sugar is also available in our specification language, which is not formally detailed in Figure 2 for clarity. For example, the field element in $\mathbb{Z}_{2^{255}-19}$ represented by `h` is `h[0]` $* 2^{51\times0} +$ `h[1]` $* 2^{51\times1} + \cdots +$ `h[4]` $* 2^{51\times4}$. That is, each element in `h` denotes a 51-bit segment (or called *limb*) of the field element. It can be written as `limbs 51 [ h[0], h[1], h[2], h[3], h[4] ]` for short, or even `limbs 51 h[0..4]`. Similarly, if one wants to specify a range property that each element `f[`i`]` $<_u 2^{51}$ for $0 \leq i \leq 4$, the shorthand `f[0..4]` $<_u 2^{51}$ can be used. Given n predicates p_i's, a useful alternative to $p_1 \land p_2 \land \cdots \land p_n$ is `and` $[p_1, p_2, \cdots, p_n]$.

Recall that users can provide extra knowledge about the target program as hints to aid the verification by attaching LLVM IR annotations. They are restricted to be of the form "`assert` *Atom* = *Atom*". The variables showing in the assertion must be variables of the target program. llvm2CryptoLine will first check whether the equation holds or not. If it holds, it is then utilized as an extra lemma to verify the postcondition. Otherwise, "FAILED" is returned.

2.3 Translation to CryptoLine

The translation from specifications in our language to CryptoLine specifications is almost straightforward. The main difficulty of translating LLVM IR programs to CryptoLine programs lies in the modeling of pointers and memory, which are not supported in the CryptoLine language. We proposed in [14] to model memory with the notion of *symbolic memory addresses*, representing memory cells symbolically. Values of pointers thus can be evaluated by symbolic execution [11]. Then the obtained symbolic values are modeled by CryptoLine variables. The translation from IR programs to untyped CryptoLine programs has been detailed in [14]. And the translation to typed CryptoLine is very similar. We do not repeat ourselves here. Instead, we only highlight some differences.

The typed CryptoLine language permits variables of different bit-widths in one program. When 64-bit and 128-bit variables coexist in the IR program, llvm2CryptoLine does not need to mimic a 128-bit instruction via several 64-bit instructions as in [14]. Instead, it models 128-bit IR instructions directly with 128-bit CryptoLine instructions. This change reduces the sizes of generated CryptoLine programs, as we can see in Section 3, hence improving verification efficiency. Another advantage introduced by typed CryptoLine is revealed for the IR instructions `trunc` and `zext`. While their source and destination types are restricted to be 64-bit or 128-bit in [14], such restrictions are not necessary in llvm2CryptoLine. These two instructions, as well as the new supported instruction `sext`, are modeled by the type casting instruction cast in typed CryptoLine. It is also worth mentioning that llvm2CryptoLine supports the translation of double pointers. Based on the observation that double pointers found in cryptographic programs are only used by `load` or `store` with no pointer arithmetic, the pointer analysis using *pointer tables* [14] is extended in llvm2CryptoLine with a structure called *pointer-to-pointer table*, which maps each double pointer to the pointer it points to, keeping track of concrete values of all double pointers during translation.

2.4 Heuristics

The CryptoLine verification tool reduces a verification problem into an algebraic problem and a range problem, then invokes an algebraic engine (computer algebra system) and a range engine (SMT solver) to solve them respectively [7, 17]. Due to the reduction algorithm and distinct reasoning abilities of two engines, users are sometimes required to add annotations via the CryptoLine instructions assert and assume to make the verification succeed. The common annotations attached in CryptoLine programs are:

$$\text{assert true } \&\& \ e_1 = e_2; \quad \text{assume } e_1 = e_2 \ \&\& \text{ true;}$$

The assertion asks CryptoLine to check the validity of $e_1 = e_2$ using the range engine. If it succeeds, $e_1 = e_2$ is passed to the algebraic engine as a lemma by the assumption; Otherwise, the verification fails. The annotations attached to the input IR program for LLVM2CRYPTOLINE (Figure 1) are in fact translated into such pairs of assert and assume.

We design and implement heuristics to insert annotations into generated CRYPTOLINE programs automatically, minimizing efforts from users. Besides the heuristics in [14], we implement new heuristics in LLVM2CRYPTOLINE. Particularly, we develop a structure called *derivation tree* to generalize the and-after-lshr heuristic in [14]. Bitwise operations are common in cryptographic programs. However, algebraic engines are not good at analyzing bitwise operations. For instance, the following pattern is used in Curve25519 in OpenSSL:

```
%x = and i64 %a 0x7FFFFFFFFFFFF
%y = lshr i64 %a 51
```

The first 64-bit instruction and assigns x with the low 51 bits of a. The second one right-shifts a by 51 bits, hence y equals the high 13 bits of a. They together perform a bit-vector splitting. After translation, the following CRYPTOLINE snippet is generated:

and x a 0x7FFFFFFFFFFFF;   uspl y tmp a 51;

The first CRYPTOLINE instruction and does the same thing as the IR instruction and. The IR instruction lshr is modeled by the CRYPTOLINE instruction uspl, which splits a at the position 51, assigning the high 13 bits to y and low 51 bits to tmp. By analysis, we know $x = tmp$. The algebraic engine needs this crucial fact, but cannot deduce it. Annotations are thus required to ask the range engine for help. Note that both x and tmp are segments *derived* from a. Derivation trees are constructed to track derivation relations introduced by bitwise operations. Our heuristics identify equality or concatenation relations between variables by analyzing these trees.

Table 1: Experimental Results

	Function	L_{IR}	L_{CL}		*Mod*			T (s)
			[14]	ours	vanilla	[14]	ours	
	OpenSSL ecp_nistp224.c							
auto (7)	felem_diff, felem_diff_128_64, felem_mul, felem_scalar, felem_square, felem_sum, widefelem_diff							≤ 8.71
tuned (5)	felem_mul_reduce	98	293	138	36	46	2	21.43
	felem_neg	43	105	73	21	13	4	0.59
	felem_reduce	64	160	103	36	19	2	1.08
	felem_square_reduce	81	242	137	36	46	10	13.87
	widefelem_scalar	31	83	28	2	4	1	2.37
	OpenSSL ecp_nistp256.c							
auto (6)	felem_diff, felem_scalar, felem_small_sum, felem_sum, smallfelem_mul, smallfelem_neg							≤ 3.54
tuned (3)	felem_shrink	65	146	134	53	42	37	1.33
	felem_small_mul	71	135	135	54	45	38	8.12
	smallfelem_square	74	217	114	10	5	5	0.88
	OpenSSL ecp_nistp521.c							
auto (10)	felem_diff64, felem_diff128, felem_neg, felem_sum64, felem_mul, felem_scalar, felem_scalar64, felem_scalar128, felem_square, felem_diff_128_64							≤ 54.08
tuned (1)	felem_reduce	138	278	266	77	54	38	2.59
	OpenSSL curve25519.c							
auto (2)	fe51_add, fe51_sub							≤ 0.18
tuned (4)	fe51_mul	122	350	156	29	16	4	4.51
	fe51_mul121666	57	120	84	19	8	8	0.95
	fe51_sq	92	256	125	19	14	4	1.85
	x25519_scalar_mult[a]	567	1262	673	131	132	37	281*
	wolfSSL fe_operations.c							
auto (3)	fe_add, fe_sub, fe_neg							≤ 0.96
tuned (4)	fe_mul	366	-	443	80	-	50	10928*
	fe_mul121666	119	-	191	80	-	50	84.40
	fe_sq	245	-	321	80	-	50	169001*
	fe_sq2	255	-	331	80	-	50	253613*
	NaCl curve25519.c							
auto (1)	fsum							0.17
tuned (5)	fdifference_backwards[b]	77	-	134	53	-	53	1.09
	fmul	111	-	149	20	-	2	10.34
	fscalar_product	52	-	73	18	-	2	0.99
	fsquare	79	-	119	18	-	2	5.78
	fmonty[ab]	63	-	78	18	-	18	26.00

[a]Only the Montgomery Ladderstep part is verified.
[b]Range verification is disabled due to a bug exposed in [7].

3 EVALUATION

LLVM2CRYPTOLINE successfully verifies 51 C implementations of arithmetic operations in cryptographic primitives. They are from three libraries: OpenSSL 3.0.5, wolfSSL 5.5.3 and NaCl 20110221. The experimental results are demonstrated in Table 1. All verification tasks are performed on an Ubuntu 20.04.3 laptop with a 2-core 3.19 GHz CPU and 8 GB RAM, except for the complicated ones (marked with "*") on a Linux server with a 28-core 2.60 GHz CPU and 220 GB RAM. Singular 4.1.1 and Boolector 3.2.2 are set as engines for CRYPTOLINE. All verified functions are divided into two categories. "auto" indicates that those functions are verified fully automatically, without human intervention. "tuned" means that human efforts are needed. Column T shows the individual verification time. For the "tuned" category, column L_{IR} gives the sizes of extracted IR programs. Columns L_{CL} compare the sizes of generated CRYPTOLINE programs from LLVM2CRYPTOLINE ("ours") and the approach in [14], respectively. Columns *Mod* are the numbers of manual modifications required to verify the functions by LLVM2CRYPTOLINE ("ours"), the work in [14], and LLVM2CRYPTOLINE without heuristics ("vanilla"), respectively. Note that the functions from wolfSSL and NaCl are not supported by [14].

The results demonstrate that the verification of 29 (56.9%) functions is fully automatic. Most (90.2%≈46/51) of the functions can be verified with a common laptop within 1 minute. We also see that LLVM2CRYPTOLINE generates significantly smaller CRYPTOLINE programs than [14], thanks to the typed CRYPTOLINE language. As for necessary human efforts, it is easy to observe that LLVM2CRYPTOLINE requires less human intervention than the work in [14]. Comparing "ours" with "vanilla" shows that our heuristics substantially reduce the needed manual modifications.

4 CONCLUSION

This paper presents LLVM2CRYPTOLINE, a cryptographic C verification tool that reduces C verification problems to CRYPTOLINE verification problems. It successfully verifies various arithmetic implementations in real-world cryptographic libraries, illustrating its applicability, usability and efficiency. Its high degree of automation makes verification as simple as the push of a button in most cases.

ACKNOWLEDGMENTS

This work is supported by Shenzhen Science and Technology Innovation Commission (JCYJ20210324094202008), the National Natural Science Foundation of China (62002228, 61836005), the Natural Science Foundation of Guangdong Province (2022A1515011458, 2022A1515010880), the National Science and Technology Council (NSTC110-2221-E-001-008-MY3, NSTC111-2221-E-001-014-MY3, NSTC111-2634-F-002-019), the Sinica Investigator Award AS-IA-109-M01, the Data Safety and Talent Cultivation Project AS-KPQ-109-DSTCP and the Intel Fast Verified Postquantum Software Project.

REFERENCES

[1] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3958)*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.). Springer, 207–228. https://doi.org/10.1007/11745853_14

[2] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[3] Marek Chalupa, Vincent Mihalkovic, Anna Rechtácková, Lukás Zaoral, and Jan Strejcek. 2022. Symbiotic 9: String Analysis and Backward Symbolic Execution with Loop Folding - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13244)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 462–467. https://doi.org/10.1007/978-3-030-99527-0_32

[4] The NaCl Developers. 2011. NaCl: Networking and Cryptography library. https://nacl.cr.yp.to/.

[5] The wolfSSL Developers. 2023. The wolfSSL Library. https://github.com/wolfSSL/wolfssl.

[6] Daniel Dietsch, Matthias Heizmann, Dominik Klumpp, Frank Schüssele, and Andreas Podelski. 2023. Ultimate Taipan and Race Detection in Ultimate - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13994)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 582–587. https://doi.org/10.1007/978-3-031-30820-8_40

[7] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2019. Signed Cryptographic Program Verification with Typed CryptoLine. 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, 1591–1606. https://doi.org/10.1145/3319535.3354199

[8] Mikhail Y. R. Gadelha, Felipe R. Monteiro, Lucas C. Cordeiro, and Denis A. Nicole. 2019. ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 11429)*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer, 209–213. https://doi.org/10.1007/978-3-030-17502-3_15

[9] Matthias Heizmann, Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele, and Andreas Podelski. 2023. Ultimate Automizer and the CommuHash Normal Form - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13994)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 577–581. https://doi.org/10.1007/978-3-031-30820-8_39

[10] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[11] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[12] Neal Koblitz. 1987. Elliptic curve cryptosystems. *Mathematics of computation* 48, 177 (1987), 203–209.

[13] Daniel Kroening and Michael Tautschnig. 2014. CBMC - C Bounded Model Checker - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 389–391. https://doi.org/10.1007/978-3-642-54862-8_26

[14] Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2019. Verifying Arithmetic in Cryptographic C Programs. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 552–564. https://doi.org/10.1109/ASE.2019.00058

[15] Victor S. Miller. 1985. Use of Elliptic Curves in Cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings (Lecture Notes in Computer Science, Vol. 218)*, Hugh C. Williams (Ed.). Springer, 417–426. https://doi.org/10.1007/3-540-39799-X_31

[16] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3935)*, Dongho Won and Seungjoo Kim (Eds.). Springer, 156–168. https://doi.org/10.1007/11734727_14

[17] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2018. Verifying Arithmetic Assembly Programs in Cryptographic Primitives (Invited Talk). In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China (LIPIcs, Vol. 118)*, Sven Schewe and Lijun Zhang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:16. https://doi.org/10.4230/LIPIcs.CONCUR.2018.4

[18] The OpenSSL Project. 2023. The OpenSSL Library. https://github.com/openssl/openssl.

[19] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[20] Cedric Richter and Heike Wehrheim. 2019. PeSCo: Predicting Sequential Combinations of Verifiers - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 11429)*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer, 229–233. https://doi.org/10.1007/978-3-030-17502-3_19

[21] Ming-Hsien Tsai, Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Bow-Yaw Wang, and Bo-Yin Yang. 2023. Certified Verification for Algebraic Abstraction. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 13966)*, Constantin Enea and Akash Lal (Eds.). Springer, 329–349. https://doi.org/10.1007/978-3-031-37709-9_16

Received 2023-05-11; accepted 2023-07-20