It's About Time: Implementing a Verified Constant-Time Library

Sunjay Cauligi[†] Brian Johannesmeyer[†] Ariana Mirian[†] Gary Soeller[†]
[†] University of California, San Diego
{scauligi, bjohanne, amirian, gsoeller}@cs.ucsd.edu

ABSTRACT

TK when rest of paper is done

Keywords

Keywords

1. INTRODUCTION

Introduction TK When rest of paper is done

2. RELATED WORK

Over the years, we have seen an increase in potential attack vectors for malicious parties. One such attack vector is side channel attacks — utlizing leaked information to undermine the security of applications and systems. Our related work is split into two main areas: a discussion on side channel attacks and mitigations as well as a discussion on constant time implementations.

Starting over 40 years ago [8, 9, 20], there has been a plethora of work on the prevalence of side channel attacks and the problem of confinement. Side channels can be used in power analysis [6, 18] to detect certain computations, or in networking and the TCP/IP protocol [4, 15, 21] and also in memory [5, 10, 11]. Moreover, there has been a extensive amount of work on timing attacks [7, 19]. Our focus is on timing attacks and memory attacks and is thus more closely related to those works.

Moverover, there has been plenty of work completed on avoiding these side-channel attacks, mainly through various programming language institutions. For example, there has been a vast array of work that looks at information flow control in order to avoid or control the covert channels for both timing and memory [3, 12, 14, 16, 17]. We decide to take a slightly different approach and attack this problem at the language level, instead of employing information flow control tactics. The work that is most closely related to us [13] also uses transformations to avoid side channel attacks in C code. The main difference in our work and theirs is that we plan to create a library so that developers do not need to write in our language if they do not desire to do so.

3. FORMALIZATION

We formalize ConstantC by first proving the constant-time properties of a new language (ConstCore), then showing the transformation from ConstantC to ConstCore.

The language ConstCore is a While-like language with highly limited control flow. Notably, there are no conditional branching instructions, and the only loop construct is a for loop with static loop bounds. We adopt the *program-counter security model* from Molnar et al. [**?**], keeping an instruction count as part of our small-step semantics (see **??**). We then show that every function in ConstCore has the following property: For every function f, there exists an instruction count f_κ such that for any input $x_1, \ldots, x_n$ to f and current instruction count κ, the instruction count κ' when f returns is exactly $\kappa + f_\kappa$. That is to say, the number of instructions executed by any function in ConstCore does not depend on the inputs to the function. ***[TODO: program-transcript model instead?]***

We then construct a transformation from ConstantC to ConstCore. Since both ConstCore and ConstantC are side effect-free languages, we simply need to prove that for every function f in ConstantC, the corresponding transformed function f_{core} returns an equivalent value given equivalent arguments.

We ensure this equivalence by keeping a "context", which is a bitmask representing the control flow the of the function at the current statement. This bitmask is either high (all 1s) or low (all 0s). Any variable assignment `x := v` in ConstantC is transformed to the following statement in ConstCore:

```
x := ((ctx & rnset) & v) | ((~(ctx & rnset) & x)
```

where `ctx` is the context bitmask and `&`, `|`, `~` represent bitwise *and*, *or*, and *not* operations respectively. The `rnset` variable is an additional bitmask used for tracking early function return and is described below. With this transformation, variables are only updated to new values if the context at the time of execution indicates that the original program control flow would have made it to that statement.

Conditional branches are transformed by executing the statements in both branches. However, before each block is executed, the context bitmask is updated with the branch condition. Thus the statement `if bexpr then s1 else s2` is transformed to:

```
b := bexpr
oldctx := ctx
ctx := oldctx & b
s1
ctx := oldctx & (~b)
s2
ctx := oldctx
```

where b is a fresh temporary variable for each instance of a branching statement. This ensures that nested conditionals still function as expected.

Return statements in CONSTANTC are dealt with by constructing two additional variables for every function: *rval* (initialized low) and *rnset* (initialized high). A return statement in CONSTANTC is translated to an assignment to *rval*, gated by the context as above. Additionally, *rnset* ("return value not set") is updated as follows:

```
rnset := rnset & (~ctx)
```

In this way, *rnset* remains high until a "return statement" is executed under an active control flow path, at which point it is set low and remains low for the rest of the function. Since all variable assignments are gated with *rnset* in addition to the context, no further variable assignments will cause updates.

We have one final problem: even if two functions execute the same number of instructions, they can take different amounts of time ***[TODO: cite]***. Our compiler currently targets Intel x86 assembly, and it is currently unknown ***[TODO: fact check]*** which instructions in this architecture are truly constant time. Our mitigation strategy is to restrict ourselves to assumed "safe" instructions, such as basic arithmetic (except division) and comparison operators, as well as simple loads, stores, and calls. Thus the constant-time nature of compiled CONSTCORE is no less secure than the set of chosen instructions.

4. IMPLEMENTATION

After we formalized the CONSTANTC and CONSTCORE languages we implemented both a compiler for CONSTANTC and constant time cryptographic code using CONSTANTC.

4.1 CONSTANTC Compiler

We implemented a compiler using OCaml and LLVM for the CONSTANTC language. Due to timing constraints, we leave the lexer and parser for future work, but otherwise have a working compiler. This allowed fast iteration on the AST with minimal code changes. There are four main parts of the CONSTANTC compiler: the *driver*, *type checker*, *transformer*, and *IR generator*.

Driver. The *driver* controls the compilation process and is the interface between the developer and the compiler. It is a standard compiler driver, so we will omit all other details about it.

Type Checker. CONSTANTC is a statically typed language. The type system is rather primitive because the only types are booleans, ints, and byte arrays. There is not a way to create new types or structs. In the future we look to add support for C structs.

Label	Supported Operations
Types	Int/Bool/ByteArr
Statements	VarDec/Assign/ArrAssign/If/For/Return
Expressions	VarExp/ArrExp/Unop/Binop/Primitive/CallExp
Unary Op	Bitwise Not
Binary Op	Plus/Minus/GT/Bitwise And/Bitwise Or

Figure 1: **Supported Language**—We show the different types, statements, expressions, and operators our language supports.

Most code requires the need to loop. CONSTANTC allows a *for* loop, but ensures that it runs a constant number of cycles. Furthermore, CONSTANTC does not allow access to byte arrays using secret data. The only way to allow byte array access is with a constant or loop value. The type checker ensures this property which guarantees array access does not depend on a secret value.

Transformer. The transformer converts the CONSTANTC AST to CONSTCORE following the rules provided in Section 3. Since CONSTCORE is built using known constant time primitives, we can generate our LLVM IR using this language, ensuring our output runs in constant time.

IR Generator. The IR generator takes a CONSTCORE program and produces LLVM IR. The primitives used in the IR are critical to the resulting code running in constant time. For this reason, all ints are 32 bits. To the best of our knowledge, all of the allowed operations run in constant time with 32 bit ints. This is not necessarily the case with 64 bits. For the same reason, each element of a byte array is 32 bits.

We have implemented functions from openssl in our language, to show that our system can be practical. We implemented ssl3_cbc_remove_padding.c. Our comparison shows up in Section 5.

4.2 Cryptographic Functions

ssl3_cbc_remove_padding We implemented the ssl3_cbc_remove_padd in our language to show that more implementations are possible. Since our language is not fully fleshed out with all the desired features, we ran into a few problems when replicating this function. The original function utilizes structs — instead, we broke up the structs into individual variables, and any mutable objects that needed to be modified were passed in and modified in a byte array. Since we do not support memory referencing or public labels, we had to hardcode the size of the arrays. Moreover, we do not support unsigned types in our language, so wherever an unsigned was used in the original function, we used an int in our code.

5. EVALUATION

We were not able to evaluate our implementation due to a variety of reasons.

[2]

Moreover, we were not able to perform microbenchmarks on this fucntion because it would not have been fair. We only implemented one function that cannot stand on it's own — in order to truly test this, we would like to implement the functinos needed to encrypt and decrypt, so that we can test the function is it's full form.

6. FUTURE WORK

Our goal for this quarter was to create a few implementations in our language to show that this concept is feasible and work further research. As such, we have a plethora of future work that we would like to explore in the months to folow. The first, which is seemingly intuitive, is that we want to implement more functions in our language, so that it becomes a full fledged library, instead of the test library as it stands now. Moreover, we need to create a syntax for our language — in it's current implementation, we have an AST, which can transform into IR. Having a corresponding syntax is key to stabilize our language.

Currently, our language has only private labels — we would like to expand it to also utilize public labels. Along these lines, we also would like to expand the notion of our side channel safety — in it's current form, our language prevents timing attacks. We would like to expand it to include memory safety and prevention against memory side channels. This includes fixing our language so that it does not index based on a secret value.

Finally, a big future goal of this project is to determine a method of usability testing for programming languages. A huge impediment in this task is that there is no good way to test programming languages — it is not a simple task that you can assign humans, such as with Amazon Mechanical Turk [1]. One needs to be able to find the key group of users, and be able to test it at a large scale in order to get data about it. This is an open question that we are hoping to address in the following months.

7. CONCLUSION

Conclusion **TK** when rest of paper is done

8. REFERENCES

[1] Amazon mechanical turk. https://www.mturk.com/mturk/welcome.

[2] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, Aug. 2016. USENIX Association.

[3] P. Buiras, A. Levy, D. Stefan, A. Russo, and D. Mazières. *A Library for Removing Cache-Based Attacks in Concurrent Information Flow Systems*, pages 199–216. Springer International Publishing, Cham, 2014.

[4] S. Cabuk, C. Brodley, and C. Shields. Ip covert timing channels: design and detection. In *CCS '04 Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004.

[5] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205, May 2013.

[6] B. Koziel, A. Jalali, R. Azarderakhsh, D. Jao, and M. Mozaffari-Kermani. *NEON-SIDH: Efficient Implementation of Supersingular Isogeny Diffie-Hellman Key Exchange Protocol on ARM*, pages 88–103. Springer International Publishing, Cham, 2016.

[7] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 324–335, July 2009.

[8] B. W. Lampson. A note on the confinment problem. In *Communications of the ACM*, volume 16, pages 613–615. ACM, 1973.

[9] S. B. Lipner. A comment on the confinement problem. In *SOSP '75 Proceedings of the fifth ACM symposium on Operating systems principles*. ACM, 1975.

[10] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622, Washington, DC, USA, 2015. IEEE Computer Society.

[11] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[12] J. C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman. Information-flow control for programming on encrypted data. In *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, 2012.

[13] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology*, ICISC'05, pages 156–168, Berlin, Heidelberg, 2006. Springer-Verlag.

[14] J. Planul and J. C. Mitchell. Oblivious program execution and path-sensitive non-interference. In *Computer Security Foundations Symposium (CSF)*. IEEE, 2013.

[15] C. H. Rowland. Covert channels in the tcp/ip protocol suite. *First Monday*, 2(5), 1997.

[16] D. Stefan, P. Buiras, E. Yang, A. Levy, D. Terei, A. Russo, and D. Mazieres. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS 2013)*, 2013.

[17] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 201–214, New York, NY, USA, 2012. ACM.

[18] S. Vaudenay. *Side-Channel Attacks on Threshold Implementations Using a Glitch Algebra*, pages 55–70. Springer International Publishing, Cham, 2016.

[19] Z. Wang and R. Lee. Covert and side channels due to processor architecture. In *ACSAC '06 Proceedings of the 22nd Annual Computer Security Applications Conference*. ACM, 2004.

[20] J. C. Wray. An analysis of covert timing channels. In *ACM Jounal of Computer Science*, volume 1, pages 219–222. ACM, 1991.

[21] S. Zander, G. Armitage, and P. Branch. A survey of covert channels and countermeasures in computer network protocols. In *IEEE Communications Surveys & Tutorials*, volume 9, pages 44–57. IEEE, 2007.