

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

Side channels are a mechanism to communicate information about a computation. Various attacks exist on side channels, including timing and power analysis attacks. Timing attacks are among the most common. This is where an adversary is able to measure the amount of time required to complete a computation. By doing so, information about the data being computed on is leaked. Among many, one notable timing attack has been shown to leak a complete AES key from a remote server [3].

Due to the sensitive data that is leaked from timing attacks, various defenses have been implemented in libraries like LIO [12]. Since libraries like this are not available in C, many developers that work on cryptographic code use a technique called constant time programming to write constant time code. The idea behind constant time programming is to structure the code so that the executable runs in constant time with respect to sensitive data. There are three main issues with constant time programming. *First*, it restricts how the developer can use high level constructs like if-statements and loops, making the code difficult to read and write. This level of complexity can easily cause the developer to write buggy code if it is not tested properly. *Second*, since the code is written in a high level language, they may be using primitives that are not actually constant time (e.g., floating point division). *Finally*, the compiler may optimize any constant time code to a form that is not actually constant time.

We address the need to easily write verified constant time code by making four contributions to constant time programming. *First*, we create a core language, built using low level primitives that are constant time. *Second*, we formalize code transformations from our high level language, CONSTANC, to our core language, CONSTCORE. *Third*, we build a compiler for CONSTANC that produces constant time libraries. *Finally*, we write cryptographic libraries in CONSTANC and verify they run in constant time with ct-verif [2].

2. FORMALIZATION

We formalize CONSTANC by first proving the constant-time properties of a new language (CONSTCORE), then showing the transformation from CONSTANC 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-transcript security model* from Molnar et al. [9], keeping a transcript of executed instructions as part of our small-step semantics (see Appendix A). We then show that every function in CONSTCORE has the following property: For every function f , there exists an transcript f_κ such that for any input x_1, \dots, x_n to f and current transcript κ , the transcript κ' when f returns is exactly $\kappa + f_\kappa$. That is to say, the program path through any function in CONSTCORE does not depend on the inputs to the function.

We then construct a transformation from CONSTANC to CONSTCORE. Since both CONSTCORE and CONSTANC are side effect-free languages, we simply need to prove that for every function f in CONSTANC, 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 of the function at the current statement. This bitmask is either *high* (all 1s) or *low* (all 0s). Any variable assignment $x := v$ in CONSTANC is transformed to the following statement in CONSTCORE:

$$x := ((\text{ctx} \& \text{rnsset}) \& v) \mid (\sim(\text{ctx} \& \text{rnsset}) \& x)$$

where ctx is the context bitmask and $\&$, \mid , \sim represent bitwise *and*, *or*, and *not* operations respectively. The formal transformation is given in rule STT-VAR-ASSIGN. The rnsset 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 $\langle s1 \rangle$ and $\langle s2 \rangle$ are the transformations of `s1` and `s2`, and `b` is a fresh temporary variable for each instance of a branching statement. This ensures that nested conditionals still function as expected. The formal transformation is given in rule STT-IF.

Return statements in CONSTANC are dealt with by constructing two additional variables for every function: *rval* (initialized low) and *rnset* (initialized high). A return statement in CONSTANC 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. The formal transformation is given in rule STT-RET.

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.

3. IMPLEMENTATION

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

3.1 CONSTANC Compiler

We implemented a compiler using OCaml and LLVM for the CONSTANC 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 CONSTANC 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

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.

a standard compiler driver, so we will omit all other details about it.

Type System. CONSTANC is a statically typed language. The type system is rather primitive because the only types are booleans, ints, and byte arrays. We guarantee type safety by type checking the CONSTANC AST. Types are checked again during IR generation, as LLVM IR is also typed. At the moment, there is not a way to create new types or structs. In the future we look to add support for C structs.

Most code requires the need to loop. CONSTANC allows a *for* loop, but ensures that it runs a constant number of cycles. Furthermore, CONSTANC does not allow access to byte arrays using secret data. The only supported ways to index into a byte array are with a constant or a loop value. The type checker ensures this property, guaranteeing that array access do not depend on secret values. The type checker also prevents *index out of bounds* errors by checking the index of the constant or loop value against the byte array size, which is known at compile time.

Transformer. The transformer converts the CONSTANC AST to CONSTCORE following the rules provided in Section 2. 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 4.

3.2 Cryptographic Functions

ssl3_cbc_remove_padding We implemented the `ssl3_cbc_remove_padding` 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.

4. 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 function because it would not have been fair. We only implemented one function that cannot stand on its own — in order to truly test this, we would like to implement the functions needed to encrypt and decrypt, so that we can test the function in its full form.

5. 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 follow. 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 its 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 its 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.

6. RELATED WORK

The security community has been aware of side channel attacks for over 40 years [5, 6, 15]. Recently, side channels have been shown to leak information via power analysis [13]. Furthermore, timing side channels have been shown to leak information from the cache [7] or other aspects of processor architecture [14].

Due to the rise in the number of side channel attacks, there has recently been a rise in defenses. [8, 10] uses information flow control to prevent information leaks on encrypted data. [11] uses instruction-based scheduling to address cache based timing attacks. [4, 12] address timing attacks in LIO, a concurrent information flow control system for Haskell. Most closely related to CONSTANC, [9] uses C to C code transformations to avoid timing side channels.

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] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.
- [4] 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.
- [5] B. W. Lampson. A note on the confinement problem. In *Communications of the ACM*, volume 16, pages 613–615. ACM, 1973.
- [6] S. B. Lipner. A comment on the confinement problem. In *SOSP '75 Proceedings of the fifth ACM symposium on Operating systems principles*. ACM, 1975.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] J. Planul and J. C. Mitchell. Oblivious program execution and path-sensitive non-interference. In *Computer Security Foundations Symposium (CSF)*. IEEE, 2013.
- [11] D. Stefan, P. Buiras, E. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS 2013)*, 2013.
- [12] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. 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.
- [13] S. Vaudenay. *Side-Channel Attacks on Threshold Implementations Using a Glitch Algebra*, pages 55–70. Springer International Publishing, Cham, 2016.
- [14] 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.
- [15] J. C. Wray. An analysis of covert timing channels. In *ACM Journal of Computer Science*, volume 1, pages 219–222. ACM, 1991.

APPENDIX

A. SEMANTICS OF `CONSTANC`

A.1 `CONSTCORE` Grammar

\ominus	$::=$ $ \sim$	unary operations bitwise not
\oplus	$::=$ $ +$ $ -$ $ *$ $ \ll$ $ \gg$ $ \&$ $ $ $ ==_s$ $!=_s$ $ >_s$ $ <_s$ $ >=_s$ $ <=_s$	binary operations bitwise and bitwise or equals (sign extended)
e	$::=$ $ \text{TRUE}$ $ \text{FALSE}$ $ c$ $ a$ $ x$ $ a[e]$ $ \ominus e$ $ e_1 \oplus e_2$ $ f(e_1, \dots, e_n)$	expressions bitmask true (0b1111...) bitmask false (0b0000...) numeric value bytearray variable array access unary operation binary operation function application
s	$::=$ $ \text{skip}$ $ s_1; s_2$ $ \text{def } x := e$ $ \text{adef } x := a$ $ x := e$ $ a[e_1] := e_2$ $ \text{for } x \text{ from } v_1 \text{ to } v_2 : s$	statements skip sequence variable declaration array declaration variable assignment array assignment for loop
$fval$	$::=$ $ (x_1, \dots, x_n) : s @ e$	function spec
$fndef$	$::=$ $ \text{fdef } fval$	function definition
$program$	$::=$ $ fndef_1; \dots; fndef_n; \text{expose } fndef$	program list of fdefs

A.2 CONSTCORE Small-Step Semantics

Λ	$::=$	function store
	\emptyset_Λ	empty function store
	$\Lambda[f \mapsto fval]$	define function
Γ	$::=$	global memory
	\emptyset_Γ	
	$\Gamma[a \mapsto []]$	new array
	$\Gamma[a \mapsto \Gamma(a)[v_1 \mapsto v_2]]$	array update
μ	$::=$	local memory
	\emptyset_μ	empty memory
	$\mu[x \mapsto v]$	add/update variable
	$\mu_1 \triangleright \mu_2$	push stack frame
$\{\sigma\}$	$::=$	variable substitution
	$\{x_1/v_1, \dots, x_k/v_k\}$	

$$\boxed{\{\Lambda, \Gamma, \mu, \kappa\} e \longrightarrow \{\Lambda', \Gamma', \mu', \kappa'\} e'} \quad (e \text{ reduces to } e')$$

$$\frac{\text{EXR-VAR} \quad \mu = \mu'[x \mapsto v] \quad \kappa' = \kappa \triangleright \mathbf{load}}{\{\Lambda, \Gamma, \mu, \kappa\} x \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} v}$$

$$\frac{\text{EXR-ARR-GET-EXPR} \quad \{\Lambda, \Gamma, \mu, \kappa\} e \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'}{\{\Lambda, \Gamma, \mu, \kappa\} a[e] \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} a[e']}$$

$$\frac{\text{EXR-ARR-GET-VAL} \quad v' = \Gamma(a)[v] \quad \kappa' = \kappa \triangleright \mathbf{load}}{\{\Lambda, \Gamma, \mu, \kappa\} a[v] \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} v'}$$

$$\frac{\text{EXR-UNOP-EXPR} \quad \{\Lambda, \Gamma, \mu, \kappa\} e \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'}{\{\Lambda, \Gamma, \mu, \kappa\} \ominus e \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} \ominus e'}$$

$$\frac{\text{EXR-UNOP-VAL} \quad v' \equiv \llbracket \ominus v \rrbracket \quad \kappa' = \kappa \triangleright \ominus}{\{\Lambda, \Gamma, \mu, \kappa\} \ominus v \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} v'}$$

$$\frac{\text{EXR-BINOP-L} \quad \{\Lambda, \Gamma, \mu, \kappa\} e_1 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'_1}{\{\Lambda, \Gamma, \mu, \kappa\} e_1 \oplus e_2 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'_1 \oplus e_2}$$

$$\frac{\text{EXR-BINOP-R} \quad \{\Lambda, \Gamma, \mu, \kappa\} e_2 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'_2}{\{\Lambda, \Gamma, \mu, \kappa\} v \oplus e_2 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} v \oplus e'_2}$$

$$\frac{\text{EXR-BINOP-VAL} \quad v_3 \equiv \llbracket v_1 \oplus v_2 \rrbracket \quad \kappa' = \kappa \triangleright \oplus}{\{\Lambda, \Gamma, \mu, \kappa\} v_1 \oplus v_2 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} v_3}$$

$$\frac{\text{EXR-SUBST-EMPTY}}{\{\Lambda, \Gamma, \mu, \kappa\} \{\} e \longrightarrow \{\Lambda, \Gamma, \mu, \kappa\} e}$$

$$\frac{\text{EXR-SUBST-EXPR} \quad \{\Lambda, \Gamma, \mu, \kappa\} e \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} e \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} \{\sigma\} e'}$$

$$\frac{\text{EXR-SUBST-VAR} \quad \kappa' = \kappa \triangleright \mathbf{load}}{\{\Lambda, \Gamma, \mu, \kappa\} \{x_1/v_1, \dots, x_k/v_k\} x_i \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} x_i}$$

$$\frac{\text{EXR-SUBST-NO-VAR}}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} x \longrightarrow \{\Lambda, \Gamma, \mu, \kappa\} x}$$

$$\frac{\text{EXR-SUBST-VAL}}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} v \longrightarrow \{\Lambda, \Gamma, \mu, \kappa\} v}$$

$$\frac{\text{EXR-FN-EXPR} \quad \{\Lambda, \Gamma, \mu, \kappa\} e_1 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'_1}{\{\Lambda, \Gamma, \mu, \kappa\} f(v_1, \dots, v_k, e_1, e_2, \dots, e_n) \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} f(v_1, \dots, v_k, e'_1, e_2, \dots, e_n)}$$

$$\frac{\text{EXR-FN-SUBST} \quad \kappa' = \kappa \triangleright \mathbf{store}}{\{\Lambda, \Gamma, \mu, \kappa\} f(x'_1/v'_1, \dots, x'_k/v'_k, v_1, v_2, \dots, v_n) \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} f(x'_1/v'_1, \dots, x'_k/v'_k, x_1/v_1, v_2, \dots, v_n)}$$

$$\frac{\text{EXR-FN-CALL} \quad \Lambda = \Lambda'[f \mapsto (x_1, \dots, x_k) : s @ e] \quad \mu' = \mu \triangleright \emptyset_\mu \quad \kappa' = \kappa \triangleright f}{\{\Lambda, \Gamma, \mu, \kappa\} f(x_1/v_1, \dots, x_k/v_k) \longrightarrow \{\Lambda, \Gamma, \mu', \kappa'\} \{x_1/v_1, \dots, x_k/v_k\} s @ e}$$

EXR-SKIP-EXPR

$$\frac{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} e \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} \mathbf{skip} @e \longrightarrow \{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} \mathbf{skip} @e'}$$

EXR-SKIP-VAL

$$\frac{\mu = \mu_1 \triangleright \mu_2 \quad \kappa' = \kappa \triangleright \mathbf{ret}}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} \mathbf{skip} @v \longrightarrow \{\Lambda, \Gamma, \mu_1, \kappa'\} v}$$

EXR-SEQ

$$\frac{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} s_1 @e_0 \longrightarrow \{\Lambda, \Gamma, \mu', \kappa'\} \{\sigma'\} s'_1 @e_0}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} s_1; s_2 @e_0 \longrightarrow \{\Lambda, \Gamma, \mu', \kappa'\} \{\sigma'\} s'_1; s_2 @e_0}$$

EXR-SEQ-SKIP

$$\frac{}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} \mathbf{skip}; s @e_0 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} s @e_0}$$

EXR-DEF-EXPR

$$\frac{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} e \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} \mathbf{def} x := e @e_0 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} \{\sigma\} \mathbf{def} x := e' @e_0}$$

EXR-DEF-VAL

$$\frac{\mu' = \mu[x \mapsto v] \quad \kappa' = \kappa \triangleright \mathbf{store}}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} \mathbf{def} x := v @e_0 \longrightarrow \{\Lambda, \Gamma, \mu', \kappa'\} \{\sigma\} \mathbf{skip} @e_0}$$

EXR-DEF-ARR

$$\frac{\Gamma' = \Gamma[a \mapsto []] \quad \mu' = \mu[x \mapsto a] \quad \kappa' = \kappa \triangleright \mathbf{store}}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} \mathbf{adef} x := a @e_0 \longrightarrow \{\Lambda, \Gamma', \mu', \kappa'\} \{\sigma\} \mathbf{skip} @e_0}$$

EXR-ASSIGN-EXPR

$$\frac{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} e \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} x := e @e_0 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} \{\sigma\} x := e' @e_0}$$

EXR-ASSIGN-VAL

$$\frac{\mu' = \mu[x \mapsto v] \quad \kappa' = \kappa \triangleright \mathbf{store}}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} x := v @e_0 \longrightarrow \{\Lambda, \Gamma, \mu', \kappa'\} \{\sigma\} \mathbf{skip} @e_0}$$

EXR-ARR-ASSIGN-EXPR-L

$$\frac{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} e_1 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'_1}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} a[e_1] := e_2 @e_0 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} \{\sigma\} a[e'_1] := e_2 @e_0}$$

EXR-ARR-ASSIGN-EXPR-R

$$\frac{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} e_2 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} e'_2}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} a[v_1] := e_2 @e_0 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} \{\sigma\} a[v_1] := e'_2 @e_0}$$

EXR-ARR-ASSIGN-VAL

$$\frac{\Gamma' = \Gamma[a \mapsto \Gamma(a)[v_1 \mapsto v_2]] \quad \kappa' = \kappa \triangleright \mathbf{store}}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} a[v_1] := v_2 @e_0 \longrightarrow \{\Lambda, \Gamma', \mu, \kappa'\} \{\sigma\} \mathbf{skip} @e_0}$$

EXR-FOR

$$\frac{v_1 < v_2 \quad v'_1 = v_1 + 1 \quad \kappa' = \kappa \triangleright \mathbf{store}}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} \mathbf{for} x \mathbf{from} v_1 \mathbf{to} v_2 : s @e_0 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} \{\sigma\} (\{x/v_1\} s); \mathbf{for} x \mathbf{from} v'_1 \mathbf{to} v_2 : s @e_0}$$

EXR-ADD-SUBST

$$\frac{\begin{array}{l} \{\sigma_1\} \cap \{\sigma_2\} = \{\} \\ \{\sigma_3\} = \{\sigma_1\} \cup \{\sigma_2\} \end{array}}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma_1\} (\{\sigma_2\} s) @e_0 \longrightarrow \{\Lambda, \Gamma, \mu, \kappa'\} \{\sigma_3\} s @e_0}$$

EXR-FOR-BASE

$$\frac{v_1 = v_2}{\{\Lambda, \Gamma, \mu, \kappa\} \{\sigma\} \mathbf{for} x \mathbf{from} v_2 \mathbf{to} v_2 : s @e_0 \longrightarrow \{\Lambda, \Gamma, \mu', \kappa\} \{\sigma\} \mathbf{skip} @e_0}$$

A.3 CONSTANC Grammar

\ominus_h	$::=$ $ $! $ $ ~	unary operations logical not bitwise not
\oplus_h	$::=$ $ $ + $ $ - $ $ * $ $ << $ $ >> $ $ & $ $ $ $ && $ $ $ $ == $ $!= $ $ > $ $ < $ $ >= $ $ <=	binary operations
e_h	$::=$ $ $ b_h $ $ c_h $ $ a $ $ x $ $ $a[e_h]$ $ $ $\ominus_h e_h$ $ $ $e_{h1} \oplus e_{h2}$ $ $ $f_h(e_{h1}, \dots, e_{hn})$	expressions boolean value numeric value bytearray variable array access unary operation binary operation function application
s_h	$::=$ $ $ skip _{h} $ $ $s_{h1}; s_{h2}$ $ $ def _{h} $x := e_h$ $ $ adef _{h} $x := a$ $ $ $x := e_h$ $ $ $a[e_{h1}] := e_{h2}$ $ $ for _{h} x from v_{h1} to v_{h2} $ $ if _{h} e_h then s_{h1} else s_{h2} $ $ return _{h} e_h	statements skip sequence variable declaration array declaration variable assignment array assignment for loop conditional branch return
$hfval$	$::=$ $ $ $(x_1, \dots, x_n) : s_h$	function spec
$hfndef$	$::=$ $ $ fdef _{h} f_h $hfval$	function definition
$hprogram$	$::=$ $ $ $hfndef_1; \dots; hfndef_n; \textbf{expose}$ $hfndef$	program list of fdefs

A.4 Transformations from CONSTANC to CONSTCORE

$\llbracket \ominus_h \rrbracket_t = \ominus$	$(\ominus_h \text{ is transformed to } \ominus)$					
UNOPT-LNOT			UNOPT-UNOP			
$\frac{}{\llbracket ! \rrbracket_t = \sim}$			$\frac{}{\llbracket \ominus_h \rrbracket_t = \ominus}$			
$\llbracket \oplus_h \rrbracket_t = \oplus$						
$(\oplus_h \text{ is transformed to } \oplus)$						
BINOPT-LAND	BINOPT-LOR	BINOPT-EQ	BINOPT-NEQ	BINOPT-GT	BINOPT-LT	BINOPT-GTE
$\frac{}{\llbracket \&\& \rrbracket_t = \&}$	$\frac{}{\llbracket \mid \mid \rrbracket_t = \mid}$	$\frac{}{\llbracket == \rrbracket_t = ==_s}$	$\frac{}{\llbracket != \rrbracket_t = !=_s}$	$\frac{}{\llbracket > \rrbracket_t = >_s}$	$\frac{}{\llbracket < \rrbracket_t = <_s}$	$\frac{}{\llbracket >= \rrbracket_t = >=_s}$
BINOPT-LTE			BINOPT-BINOP			
$\frac{}{\llbracket <= \rrbracket_t = <=_s}$			$\frac{}{\llbracket \oplus_h \rrbracket_t = \oplus}$			
$\llbracket e_h \rrbracket_t = e$						
$(e_h \text{ is transformed to } e)$						
EXT-VAL	EXT-VAR	EXT-ARR	EXT-ARR-GET	EXT-FN-CALL		
$\frac{v \equiv \llbracket v_h \rrbracket_{int}}{\llbracket v_h \rrbracket_t = v}$	$\frac{}{\llbracket x \rrbracket_t = x}$	$\frac{}{\llbracket a \rrbracket_t = a}$	$\frac{}{\llbracket e_h \rrbracket_t = e}$	$\frac{\llbracket \mathbf{fdef}_h f_h hfval \rrbracket_t = \mathbf{fdef} f fval}{\llbracket e_{h1} \rrbracket_t = e_1 \quad \dots \quad \llbracket e_{hk} \rrbracket_t = e_k}$		
				$\frac{}{\llbracket f_h(e_{h1}, \dots, e_{hk}) \rrbracket_t = f(e_1, \dots, e_k)}$		
$\llbracket s_h \rrbracket_{ctx} = s$						
$(s_h \text{ is transformed to } s)$						
STT-SKIP	STT-SEQ		STT-VAR-DEC			
$\frac{}{\llbracket \mathbf{skip}_h \rrbracket_{ctx} = \mathbf{skip}}$	$\frac{\llbracket s_{h1} \rrbracket_{ctx} = s_1 \quad \llbracket s_{h2} \rrbracket_{ctx} = s_2}{\llbracket s_{h1}; s_{h2} \rrbracket_{ctx} = s_1; s_2}$		$\frac{}{\llbracket e_h \rrbracket_t = e}$			
			$\frac{}{\llbracket \mathbf{def}_h x := e_h \rrbracket_{ctx} = \mathbf{def} x := e}$			
STT-ARR-DEC			STT-ARR-ASSIGN			
$\frac{}{\llbracket \mathbf{adef}_h x := a \rrbracket_{ctx} = \mathbf{adef} x := a}$			$\frac{\llbracket e_{h1} \rrbracket_t = e_1}{\llbracket e_{h2} \rrbracket_t = e_2 \quad e' = ctx \& rnset}$			
			$\frac{e'' = e_2 \& e' \quad e''' = x \& (\sim e')}{\llbracket x := e_h \rrbracket_{ctx} = x := (e'' \mid e''')}$			
			$\frac{e'' = e_2 \& e' \quad e''' = a[e_1] \& (\sim e')}{\llbracket a[e_{h1}] := e_{h2} \rrbracket_{ctx} = a[e_1] := (e'' \mid e''')}$			
STT-FOR	STT-IF					
$\frac{\llbracket v_{h1} \rrbracket_t = v_1 \quad \llbracket v_{h2} \rrbracket_t = v_2 \quad \llbracket s_h \rrbracket_{ctx} = s}{\llbracket \mathbf{for}_h x \mathbf{from} v_{h1} \mathbf{to} v_{h2} \rrbracket_{ctx} = \mathbf{for} x \mathbf{from} v_1 \mathbf{to} v_2 : s}$			$\frac{\llbracket e_h \rrbracket_t = e \quad \llbracket s_{h1} \rrbracket_{(ctx' \& ctx)} = s_1 \quad \llbracket s_{h2} \rrbracket_{(ctx' \& ctx)} = s_2}{\llbracket \mathbf{if}_h e_h \mathbf{then} s_{h1} \mathbf{else} s_{h2} \rrbracket_{ctx} = \mathbf{def} ctx' := e; s_1; ctx' := (\sim ctx'); s_2}$			
STT-RET						
$\frac{\llbracket e_h \rrbracket_t = e \quad e' = e \& (ctx \& rnset)}{\llbracket \mathbf{return}_h e_h \rrbracket_{ctx} = rval := (e' \mid rval); rnset := (rnset \& (\sim ctx))}$						
$\llbracket hfndef \rrbracket_t = fndef$						
$(hfndef \text{ is transformed to } fndef)$						
FDEFT-FDEF						
$\frac{\llbracket s_h \rrbracket_{\text{TRUE}} = s}{\llbracket \mathbf{fdef}_h f_h(x_1, \dots, x_k) : s_h \rrbracket_t = \mathbf{fdef}(x_1, \dots, x_k) : \mathbf{def} rval := \text{FALSE}; \mathbf{def} rnset := \text{TRUE}; s @ rval}$						