It's About Time: A Language-Based Approach to Constant-Time Programming

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 instead. 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 formalize 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. LANGUAGE DESIGN

The ConstanC system is comprised of two distinct languages: ConstanC proper, which end-users will develop programs in, and ConstCore, an internal language with specific timing guarantees.

2.1 ConstanC

The ConstanC language is the language we intend end-users of our system to program in. As ConstanC is intended to replace low-level code, the language is designed to feel very similar to C, to make it more accessible for developers.

There are, however, some necessary restrictions on what can be expressed in ConstanC. For example, recursion of any sort is disallowed. Loop constructs are restricted to for-loops, and loop bounds can only be constants. While these would be onerous restrictions for a general programming language, ConstanC is intended for a very specific domain where these features of a language are generally unnecessary.

The language supports 32-bit signed integers, boolean values, and arrays of bytes. Most standard operations on these values are supported.

We do not prove timing guarantees directly in ConstanC; instead, we show that the ConstCore language has the security guarantees we require, and transform all programs in ConstanC to functionally equivalent programs in ConstCore.

The full grammar of ConstanC can be found in Appendix A.3.

2.2 ConstCore

The ConstCore language 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. The type system is also more restrictive than that of ConstanC: we support only 32-bit signed integers and byte-arrays.

Comparison operators in ConstCore evaluate integer-width bitmasks of either all high bits (if the comparison is true) or all low bits (if false). The resulting bitmasks are then typically used to mask expressions that depend on the conditional result. For example, if a variable `x` was to be

$$\textsc{Stt-var-assign}\quad \frac{[\![e_h]\!]_t = e \qquad e' = ctx \,\&\, rnset \qquad e'' = e \,\&\, e' \qquad e''' = x \,\&\, (\sim e')}{[\![x := e_h]\!]_{ctx} = x := (e'' \mid e''')}$$

$$\textsc{Stt-if}\quad \frac{[\![e_h]\!]_t = e \qquad [\![s_{h1}]\!]_{(ctx' \,\&\, ctx)} = s_1 \qquad [\![s_{h2}]\!]_{(ctx' \,\&\, ctx)} = s_2}{[\![\textbf{if}_h\ e_h\ \textbf{then}\ s_{h1}\ \textbf{else}\ s_{h2}]\!]_{ctx} = \textbf{def}\, ctx' := e; s_1; ctx' := (\sim ctx'); s_2}$$

$$\textsc{Stt-ret}\quad \frac{[\![e_h]\!]_t = e \qquad e' = e \,\&\, (ctx \,\&\, rnset)}{[\![\textbf{return}_h\ e_h]\!]_{ctx} = rval := (e' \mid rval); rnset := (rnset \,\&\, (\sim ctx))}$$

Figure 1: Excerpt of semantic transformations from ConstantC to ConstCore. Full rule set can be found in Appendix A.4.

incremented by 3 only if the variable y was greater than 5, we could express that as the following statement:

```
x := x + ((y > 5) & 3)
```

The result of the expression `y > 5` will either be `0b1111...` if true or `0b0000...` if false. The `&` operator performs a bitwise masking operation, either leaving the value 3 as-is (if the bitmask was all 1s) or zeroing it out (if the bitmask was all 0s). Thus x gets incremented only if the condition holds.

To prove the constant-time nature of ConstCore, we adopt the *program-transcript security model* from Molnar et al. [9], keeping a log 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 $x_1, \ldots, x_n$ 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.

However, even if a function always follows the same program transcript, it might still take varying amounts of real-world time, depending on how the physical CPU implements its instructions ***[TODO: cite]***. Our compiler currently targets LLVM IR, but with Intel x86 assembly in mind; it is currently unknown ***[TODO: fact check]*** which instructions in this architecture are truly constant time. However, certain instructions, such as integer division (`idiv`), are known to have timing variations based on the input values ***[TODO: cite]***. Our mitigation strategy is to restrict ourselves to instructions that, to the best of our knowledge, run in constant time with respect to arguments. These include basic arithmetic (excluding division and modulo) and comparison operators, as well as simple loads, stores, and calls. The constant-time nature of compiled ConstCore is thus reduced to the timing properties of the set of chosen instructions.

2.3 Transformations

If we can transform any program in ConstantC to an equivalent program in ConstCore, then we can compile any ConstantC program to a program with the timing guarantees proven in 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 in ConstantC is transformed via the semantic transformation rule Stt-var-assign shown in Figure 1, 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 is transformed via the rule Stt-if in Figure 1, where ⟨s1⟩ and ⟨s2⟩ 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.

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 to reflect the current return status. 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 can be seen in the rule Stt-ret in Figure 1.

3. IMPLEMENTATION

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

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 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 know at compile time.

Transformer. The transformer converts the CONSTANC AST to CONSTCORE following the rules provided in Section **??**. 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.c 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.

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

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 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.

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 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.

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

In this paper, we introduced CONSTANC, a high level language that can be used for constant time programming. We formalized the semantics of CONSTANC and CONSTCORE and the transformation from CONSTANC to CONSTCORE. With this formalization, we built a CONSTANC compiler which we successfully used to write various cryptographic functions. Our evaluation is limited to the success of writing these functions. In the future, we plan to evaluate CONSTANC based on performance and usability. Furthermore, we plan to create a syntax for CONSTANC and implement a lexer and parser for the compiler. Finally, we plan to add public labels to the type system allowing for more flexibility for the developer and better performance.

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 confinment 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. 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.

[12] 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.

[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 Jounal of Computer Science*, volume 1, pages 219–222. ACM, 1991.

APPENDIX

A. SEMANTICS OF CONSTANC

A.1 CONSTCORE Grammar

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

A.2 CONSTCORE Small-Step Semantics

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

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

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

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

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

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

EXR-UNOP-VAL
$$\frac{v' \equiv [\![\ominus v]\!] \qquad \kappa' = \kappa \triangleright \ominus}{\{\Lambda,\Gamma,\mu,\kappa\}\, \ominus v \longrightarrow \{\Lambda,\Gamma,\mu,\kappa'\}\, v'}$$

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

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

EXR-BINOP-VAL
$$\frac{v_3 \equiv [\![v_1 \oplus v_2]\!] \qquad \kappa' = \kappa \triangleright \oplus}{\{\Lambda,\Gamma,\mu,\kappa\}\, v_1 \oplus v_2 \longrightarrow \{\Lambda,\Gamma,\mu,\kappa'\}\, v_3}$$

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

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

EXR-SUBST-VAR
$$\frac{\kappa' = \kappa \triangleright \textbf{load}}{\{\Lambda,\Gamma,\mu,\kappa\}\, \{x_1/v_1, \ldots, x_k/v_k\}\, x_i \longrightarrow \{\Lambda,\Gamma,\mu,\kappa'\}\, v_i}$$

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

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

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

EXR-FN-SUBST
$$\frac{\kappa' = \kappa \triangleright \textbf{store}}{\{\Lambda,\Gamma,\mu,\kappa\} f(x_1'/v_1', .., x_k'/v_k', v_1, v_2, .., v_n) \longrightarrow \{\Lambda,\Gamma,\mu,\kappa'\} f(x_1'/v_1', .., x_k'/v_k', x_1/v_1, v_2, .., v_n)}$$

EXR-FN-CALL
$$\frac{\begin{array}{c}\Lambda = \Lambda'[f \mapsto (x_1, .., x_k) : s \;@ e] \\ \mu' = \mu \triangleright \emptyset_\mu \qquad \kappa' = \kappa \triangleright f\end{array}}{\{\Lambda,\Gamma,\mu,\kappa\} f(x_1/v_1, .., x_k/v_k) \longrightarrow \{\Lambda,\Gamma,\mu',\kappa'\}\, \{x_1/v_1, .., x_k/v_k\}\, s \;@ e}$$

EXR-SKIP-EXPR

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

EXR-SKIP-VAL

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

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

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

EXR-DEF-EXPR

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

EXR-DEF-VAL

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

EXR-DEF-ARR

$$\frac{\begin{array}{c}\Gamma' = \Gamma[a \mapsto [\,]] \\ \mu' = \mu[x \mapsto a] \qquad \kappa' = \kappa \triangleright \textbf{store}\end{array}}{\{\Lambda,\Gamma,\mu,\kappa\}\{\sigma\}\textbf{adef}\,x := a\ @e_0 \longrightarrow \{\Lambda,\Gamma',\mu',\kappa'\}\{\sigma\}\textbf{skip}\ @e_0}$$

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

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

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

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

EXR-FOR

$$\frac{\begin{array}{c}v_1 < v_2 \\ v_1' = v_1 + 1 \qquad \kappa' = \kappa \triangleright \textbf{store}\end{array}}{\{\Lambda,\Gamma,\mu,\kappa\}\{\sigma\}\textbf{for}\,x\,\textbf{from}\,v_1\,\textbf{to}\,v_2 : s\ @e_0 \longrightarrow \{\Lambda,\Gamma,\mu,\kappa'\}\{\sigma\}(\{x/v_1\}s);\textbf{for}\,x\,\textbf{from}\,v_1'\,\textbf{to}\,v_2 : s\ @e_0}$$

EXR-ADD-SUBST

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

EXR-FOR-BASE

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

A.3 CONSTANC Grammar

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

A.4 Transformations from CONSTANC to CONSTCORE

$\boxed{[\![\ominus_h]\!]_t = \ominus}$ *($\ominus_h$ is transformed to $\ominus$)*

UNOPT-LNOT
$$\overline{[\![\texttt{!}]\!]_t = \texttt{~}}$$

UNOPT-UNOP
$$\overline{[\![\ominus_h]\!]_t = \ominus}$$

$\boxed{[\![\oplus_h]\!]_t = \oplus}$ *($\oplus_h$ is transformed to $\oplus$)*

BINOPT-LAND
$$\overline{[\![\texttt{\&\&}]\!]_t = \texttt{\&}}$$

BINOPT-LOR
$$\overline{[\![\texttt{||}]\!]_t = \texttt{|}}$$

BINOPT-EQ
$$\overline{[\![\texttt{==}]\!]_t = \texttt{==}_\texttt{s}}$$

BINOPT-NEQ
$$\overline{[\![\texttt{!=}]\!]_t = \texttt{!=}_\texttt{s}}$$

BINOPT-GT
$$\overline{[\![\texttt{>}]\!]_t = \texttt{>}_\texttt{s}}$$

BINOPT-LT
$$\overline{[\![\texttt{<}]\!]_t = \texttt{<}_\texttt{s}}$$

BINOPT-GTE
$$\overline{[\![\texttt{>=}]\!]_t = \texttt{>=}_\texttt{s}}$$

BINOPT-LTE
$$\overline{[\![\texttt{<=}]\!]_t = \texttt{<=}_\texttt{s}}$$

BINOPT-BINOP
$$\overline{[\![\oplus_h]\!]_t = \oplus}$$

$\boxed{[\![e_h]\!]_t = e}$ *(e_h is transformed to e)*

EXT-VAL
$$\frac{v \equiv [\![v_h]\!]_{int}}{[\![v_h]\!]_t = v}$$

EXT-VAR
$$\overline{[\![x]\!]_t = x}$$

EXT-ARR
$$\overline{[\![a]\!]_t = a}$$

EXT-ARR-GET
$$\frac{[\![e_h]\!]_t = e}{[\![a[e_h]]\!]_t = a[e]}$$

EXT-FN-CALL
$$\frac{[\![\mathbf{fdef}_h\ f_h\ hfval]\!]_t = \mathbf{fdef} f\, fval \quad [\![e_{h1}]\!]_t = e_1 \quad .. \quad [\![e_{hk}]\!]_t = e_k}{[\![f_h(e_{h1}, .., e_{hk})]\!]_t = f(e_1, .., e_k)}$$

$\boxed{[\![s_h]\!]_{ctx} = s}$ *(s_h is transformed to s)*

STT-SKIP
$$\overline{[\![\mathbf{skip}_h]\!]_{ctx} = \mathbf{skip}}$$

STT-SEQ
$$\frac{[\![s_{h1}]\!]_{ctx} = s_1 \qquad [\![s_{h2}]\!]_{ctx} = s_2}{[\![s_{h1}; s_{h2}]\!]_{ctx} = s_1; s_2}$$

STT-VAR-DEC
$$\frac{[\![e_h]\!]_t = e}{[\![\mathbf{def}_h\ x := e_h]\!]_{ctx} = \mathbf{def}\, x := e}$$

STT-ARR-DEC
$$\overline{[\![\mathbf{adef}_h\ x := a]\!]_{ctx} = \mathbf{adef}\, x := a}$$

STT-VAR-ASSIGN
$$\frac{[\![e_h]\!]_t = e \qquad e' = ctx \,\&\, rnset \qquad e'' = e \,\&\, e' \qquad e''' = x \,\&\, (\texttt{~}e')}{[\![x := e_h]\!]_{ctx} = x := (e'' \mid e''')}$$

STT-ARR-ASSIGN
$$\frac{[\![e_{h1}]\!]_t = e_1 \quad [\![e_{h2}]\!]_t = e_2 \qquad e' = ctx \,\&\, rnset \qquad e'' = e_2 \,\&\, e' \qquad e''' = a[e_1] \,\&\, (\texttt{~}e')}{[\![a[e_{h1}] := e_{h2}]\!]_{ctx} = a[e_1] := (e'' \mid e''')}$$

STT-FOR
$$\frac{[\![v_{h1}]\!]_t = v_1 \quad [\![v_{h2}]\!]_t = v_2 \qquad [\![s_h]\!]_{ctx} = s}{[\![\mathbf{for}_h\ x\ \mathbf{from}\ v_{h1}\ \mathbf{to}\ v_{h2}]\!]_{ctx} = \mathbf{for}\, x\, \mathbf{from}\, v_1\, \mathbf{to}\, v_2 : s}$$

STT-IF
$$\frac{[\![e_h]\!]_t = e \qquad [\![s_{h1}]\!]_{(ctx' \,\&\, ctx)} = s_1 \quad [\![s_{h2}]\!]_{(ctx' \,\&\, ctx)} = s_2}{[\![\mathbf{if}_h\ e_h\ \mathbf{then}\ s_{h1}\ \mathbf{else}\ s_{h2}]\!]_{ctx} = \mathbf{def}\, ctx' := e; s_1; ctx' := (\texttt{~}ctx'); s_2}$$

STT-RET
$$\frac{[\![e_h]\!]_t = e \qquad e' = e \,\&\, (ctx \,\&\, rnset)}{[\![\mathbf{return}_h\ e_h]\!]_{ctx} = rval := (e' \mid rval); rnset := (rnset \,\&\, (\texttt{~}ctx))}$$

$\boxed{[\![hfndef]\!]_t = fndef}$ *(hfndef is transformed to fndef)*

FDEFT-FDEF
$$\frac{[\![s_h]\!]_{\text{TRUE}} = s}{[\![\mathbf{fdef}_h\ f_h\ (x_1, .., x_k) : s_h]\!]_t = \mathbf{fdef} f\,(x_1, .., x_k) : \mathbf{def}\, rval := \text{FALSE}; \mathbf{def}\, rnset := \text{TRUE}; s\ @ rval}$$