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

In this paper we present CONSTANC: a domain specific language for writing constant-time code. We describe the two main aspects of CONSTANC: the high- and low-level languages. The high-level language includes general-purpose programming constructs such as conditionals and loops, while the low-level language is built off of constant-time primitives. We formalize a translation between the two languages which preserves functional equivalence. We then describe the compiler we built for CONSTANC. Using the CONSTANC compiler, we implement various cryptographic functions and verify they run in constant time.

1. INTRODUCTION

Timing side channels are an important class of security bugs whereby an attacker is able use the timing of events in a system to discern secret data. One well-known timing attack was the "Lucky 13" attack against various SSL implementations [1], which used minute differences in the duration between a malicious request to a server and the resulting error message to recover an SSL session key.

Due to the sensitive nature of data that can be leaked from timing attacks, developers must take extra care to defend against such attacks. The current state-of-the-art used in many cryptographic libraries is constant-time programming. The idea behind constant-time programming is to structure code so that the time a program takes to run is not influenced by 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 [1]. Second, since the code is written in a high level language, a developer may be using operations that are not actually constant time (e.g. floating point division). Finally, the compiler may optimize constant-time code to form a program that is not actually constant time.

We address the need to easily write verified constant-time code by making several contributions to constant-time programming: We formalize a core language, CONSTCORE, built using constant-time low-level primitives; we formalize code transformations from our high-level language, CONSTANC, to our core language; we build a compiler for CONSTANC that produces constant-time functions that can be linked into and called from C code; and finally, we implement cryptographic functions in CONSTANC and verify they run in constant time with *ct-verif* tool [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. Before diving into the languages, we briefly outline our threat model.

2.1 Threat Model

For CONSTANC, we assume an adversary has remote access to the machine running the code. The adversary has the ability to measure the amount of time it takes for CONSTANC code to run. However, we assume the calling program has not been compromised and that it is invoking CONSTANC functions using valid parameters.

2.2 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 where 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; a survey of typical functions indicates these language features are generally unnecessary. As we develop the language, we expect to relax some of these restrictions.

The language supports 32-bit signed integers, boolean values, and arrays of bytes. Most arithmetic and bitwise 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

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

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

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

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

in CONSTANC to functionally equivalent programs in CONSTCORE.

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

2.3 ConstCore

The CONSTCORE language is a WHILE-like language [11] 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 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 `1`s) or zeroing it out (if the bitmask was all `0`s). Thus `x` gets incremented only if the condition holds.

To show the constant-time nature of CONSTCORE, we adopt the *program-transcript security model* from Molnar et al. [11], 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 a 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, as physical instructions themselves may have timing variations depending on input registers [6]. Our compiler currently targets LLVM IR, but with Intel x86 assembly in mind; it is currently difficult which instructions in this architecture are truly constant time without support from Intel. However, certain instructions, such as integer division (`idiv`), are known to have timing variations based on the input values [5]. 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.

2.4 Transformations

If we can transform any program in CONSTANC to an equivalent program in CONSTCORE, then we can compile any CONSTANC program to a program with the timing guarantees shown in CONSTCORE.

The only possible side effect of any CONSTCORE or CONSTANC function is memory access. Thus we need to show that for every function f in CONSTANC, the corresponding transformed function f_{core} returns an equivalent value and sets equivalent memory values when given equivalent arguments and starting memory state.

We ensure this equivalence by keeping track of 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 in CONSTANC 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. 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. COMPILER

We implemented a compiler for the CONSTANTC language in OCaml, targeting LLVM bytecode. As syntax design is largely unimportant, we left the lexer and parser for future work. This allowed for fast iteration on the AST with minimal code changes. There are three major parts to the CONSTANTC compiler: *type system*, *transformer*, and *IR generator*.

Type System. Although CONSTANTC is a statically typed language, the type system is rather primitive: the only types are booleans, ints, and byte arrays. We guarantee type safety by type checking the CONSTANTC AST. Types are checked again during IR generation, as LLVM IR is also typed. At the moment, we do not have support for structs or other record types; we leave this for future work.

CONSTANTC supports a *for* loop, but ensures that it runs a constant number of cycles. Furthermore, CONSTANTC only allows byte arrays to be indexed by constants or loop indices. The type checker ensures this property, guaranteeing that array accesses do not depend on runtime values. The type checker also prevents out-of-bounds errors by checking the index against the byte array size, which can be done at compile time.

Transformer. The transformer converts the CONSTANTC AST to CONSTCORE following the rules formally defined in Appendix A.4. 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. To the best of our knowledge, all of the allowed operations run in constant time with 32 bit integers.

4. EVALUATION

Methodology In order to evaluate the correctness of CONSTANTC we used *ct-verif* [2], a contant-time verification tool that leverages the SMACK and Boogie tools to verify LLVM intermediate representation. Since CONSTANTC was built on LLVM 3.9 and *ct-verif* requires LLVM 3.5, the IR CONSTANTC produced needed to be slightly modified in order to be LLVM 3.5-compliant and hence be verified with *ct-verif*. In particular, the `load` and `getelementpr` instructions output by CONSTANTC—whose syntax is given in Figure 2—were modified to be LLVM 3.5-compliant.

LLVM	Instruction Syntax
3.5	load <ty>* <pointer>
3.9	load <ty>, <ty>* <pointer>
3.5	getelementptr <ptr vector> ptrval, <index type> idx
3.9	getelementptr <ty>, <ptr vector> <ptrval>, <index type> <idx>

Figure 2: Instructions produced by CONSTANTC with syntax changes from LLVM 3.5 to LLVM 3.9.

Test Code OpenSSL's `ssl3_cbc_remove_padding` function—given in Figure 3a—removes padding from a decrypted SSLv3 CBC record by updating the record's length in constant time. It provides an extensive yet concise test case for CONSTANTC since it encapsulates many C language features, such as structs, branches, and memory references; and constant-time features such as a notion of public and private data. In `ssl3_cbc_remove_padding`, inputs `rec->length`, `rec->data`, `block_size`, and `mac_size` are private while `s` and `rec->type` are public. Note that the branch in Figure 3a does not violate the constant-time property because its condition is based on public values.

The intuitive version of this code is given in Figure 3b, without OpenSSL's constant-time routines that avoid branching. Routines `constant_time_ge` and `constant_time_select_int` compute in constant time a greater-than-or-equal-to condition and a conditional selection, respectively. Figure 3b is truer to the programmer's original intended semantics of `ssl3_cbc_remove_padding`, without muddying its structure in order to be constant-time. The colored lines of code in Figure 3 correspond to code that is modified in this transformation into a more intuitive structure.

Limitations There are several limitations of CONSTANTC that will be resolved in future work (Section 5) and these limitations affect how much of `ssl3_cbc_remove_padding` was expressable in CONSTANTC. First, CONSTANTC does not yet support C structs; thus, `rec`'s fields were passed in via elements of a byte array. Second, CONSTANTC does not yet have any notion of public labels, so all data is conservatively assumed to be private; since memory cannot be referenced by a non-constant value, the expression `rec->data[rec->length-1]` could yet be represented without a workaround. Next, CONSTANTC currently only supports "if-then-else" and not "if" statements; an "if" statement must be represented as "if-then-else" with a dummy "else" branch. As a result, Figure 3b was altered slightly in order to be able to be represented in CONSTANTC.

Result As we don't yet have a syntax for CONSTANTC, we wrote the altered `ssl3_cbc_remove_padding` directly in CONSTANTC AST. After compilation, we verified using *ct-verif* that the generated LLVM IR was constant time.

```
int ssl3_cbc_remove_padding(const SSL *s,
  SSL3_RECORD *rec, unsigned block_size,
  unsigned mac_size) {

  unsigned padding_length, good;
  const unsigned overhead = 1 + mac_size;

  if (overhead > rec->length) return 0;
  padding_length = rec->data[rec->length - 1];

  good = constant_time_ge(
    rec->length, padding_length + overhead);
  good &= constant_time_ge(
    block_size, padding_length + 1);

  padding_length = good & (padding_length + 1);

  rec->length -= padding_length;
  rec->type |= padding_length << 8;

  return constant_time_select_int(good, 1, -1);
}
```

(a) The original constant-time OpenSSL function written in C.

```
int ssl3_cbc_remove_padding(const SSL *s,
  SSL3_RECORD *rec, unsigned block_size,
  unsigned mac_size){

  unsigned padding_length;
  const unsigned overhead = 1 + mac_size;

  if (overhead > rec->length) return 0;
  padding_length = rec->data[rec->length - 1];

  if(rec->length < padding_length + overhead)
    return -1;
  if(block_size < padding_length + 1)
    return -1;

  padding_length += 1;

  rec->length -= padding_length;
  rec->type |= padding_length << 8;

  return 1;
}
```

(b) The OpenSSL function expressed more intuitively, without using its constant-time branch-avoiding routines.

Figure 3: OpenSSL function used to evaluate CONSTANC.

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 worth further research. As such, we have a plethora of future work that we would like to explore in the months to follow. The simplest to tackle is the creation of a syntax for our language — currently we must write programs directly in the AST. Having a corresponding syntax is key to stabilizing our language.

Another useful feature will be the concept of public and private labels on data. Our language currently treats all values as private — we would like to augment it to be able to relax restrictions on data that does not need protection which allow for more programmer flexibility.

Finally, an important goal of this project is to determine a method of usability testing for programming languages. A huge impediment in this task is that there are currently no meaningful ways of user testing programming languages at scale. One needs to be able to find a key group of users and be able to test in-depth at a large scale in order to get proper data. 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 [3, 7, 8, 17]. Side channel attacks have been shown to leak information from a variety of covert channels including power analysis [15], cache access patterns [9], and other aspects of processor architecture [16].

There has much work in creating defenses against such attacks. [10, 12] use information flow control to prevent information leaks on encrypted data. [13] uses instruction-based scheduling to address cache based timing attacks. [4, 14] address timing attacks in LIO, a concurrent information flow control system for Haskell.

Most closely related to CONSTANC, Molnar et. al. use C to C code transformations to avoid timing side channels [11]. Since they do not control assembly generation, their work is only useful when using the Intel C compiler as described in their paper: they found that the GCC compiler would rewrite the `!` operator in terms of a `jmp` instruction, breaking their constant-time guarantees.

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. 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] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540. IEEE, 2013.

[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] Intel. Intel 64 and ia-32 architectures optimization reference manual. *Intel Corporation, June*, 2016.

[6] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015. In *International Conference on Cryptology and Network Security*, pages 573–582. Springer, 2016.

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

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

[17] 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
	|	`+`	integer addition
	|	`-`	integer subtraction
	|	`*`	integer multiplication
	|	`<<`	bitshift left
	|	`>>`	bitshift right
	|	`&`	bitwise and
	|	`|`	bitwise or
	|	$\texttt{==}_\texttt{s}$	equals (sign extended)
	|	$\texttt{!=}_\texttt{s}$	not equals (sign extended)
	|	$\texttt{>}_\texttt{s}$	greater than (sign extended)
	|	$\texttt{<}_\texttt{s}$	less than (sign extended)
	|	$\texttt{>=}_\texttt{s}$	greater than or equal (sign extended)
	|	$\texttt{<=}_\texttt{s}$	less than or equal (sign extended)
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

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

A.3 CONSTANC Grammar

$\ominus_h$	::=		unary operations
	|	`!`	logical not
	|	`~`	bitwise not
$\oplus_h$	::=		binary operations
	|	`+`	integer addition
	|	`-`	integer subtraction
	|	`*`	integer multiplication
	|	`<<`	bitshift left
	|	`>>`	bitshift right
	|	`&`	bitwise and
	|	`|`	bitwise or
	|	`&&`	logical and
	|	`||`	logical or
	|	`==`	equals
	|	`!=`	not equals
	|	`>`	greater than
	|	`<`	less than
	|	`>=`	greater than or equals
	|	`<=`	less than or equals
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
	|	$\mathbf{skip}_h$	skip
	|	$s_{h1}; s_{h2}$	sequence
	|	$\mathbf{def}_h\ x := e_h$	variable declaration
	|	$\mathbf{adef}_h\ x := a$	array declaration
	|	$x := e_h$	variable assignment
	|	$a[e_{h1}] := e_{h2}$	array assignment
	|	$\mathbf{for}_h\ x\ \mathbf{from}\ v_{h1}\ \mathbf{to}\ v_{h2}$	for loop
	|	$\mathbf{if}_h\ e_h\ \mathbf{then}\ s_{h1}\ \mathbf{else}\ s_{h2}$	conditional branch
	|	$\mathbf{return}_h\ e_h$	return
hfval	::=		function spec
	|	$(x_1, .., x_n) : s_h$	
hfndef	::=		function definition
	|	$\mathbf{fdef}_h\ f_h\ hfval$	
hprogram	::=		program
	|	$hfndef_1; ..; hfndef_n; \mathbf{expose}\, hfndef$	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
$$\frac{[\![\textbf{fdef}_h\ f_h\ hfval]\!]_t = \textbf{fdef}\, f\ fval \quad [\![e_{h1}]\!]_t = e_1 \quad .. \quad [\![e_{hk}]\!]_t = e_k}{[\![f_h(e_{h1}, .., e_{hk})]\!]_t = f(e_1, .., e_k)}$$

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

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

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
$$\frac{[\![e_h]\!]_t = e}{[\![\textbf{def}_h\ x := e_h]\!]_{ctx} = \textbf{def}\, x := e}$$

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

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

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

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

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

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

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