Verifying Constant-Time Implementations

José Bacelar Almeida
HASLab - INESC TEC & Univ. Minho

Manuel Barbosa
HASLab - INESC TEC & DCC FCUP

Gilles Barthe
IMDEA Software Institute

François Dupressoir
IMDEA Software Institute

Michael Emmi
Bell Labs, Nokia

Abstract

The constant-time programming discipline is an effective countermeasure against timing attacks, which can lead to complete breaks of otherwise secure systems. However, adhering to constant-time programming is hard on its own, and extremely hard under additional efficiency and legacy constraints. This makes automated verification of constant-time code an essential component for building secure software.

We propose a novel approach for verifying constant-time security of real-world code. Our approach is able to validate implementations that locally and intentionally violate the constant-time policy, when such violations are benign and leak no more information than the public outputs of the computation. Such implementations, which are used in cryptographic libraries to obtain important speedups or to comply with legacy APIs, would be declared insecure by all prior solutions.

We implement our approach in a publicly available, cross-platform, and fully automated prototype, ct-verif, that leverages the SMACK and Boogie tools and verifies optimized LLVM implementations. We present verification results obtained over a wide range of constant-time components from the NaCl, OpenSSL, FourQ and other off-the-shelf libraries. The diversity and scale of our examples, as well as the fact that we deal with top-level APIs rather than being limited to low-level leaf functions, distinguishes ct-verif from prior tools.

Our approach is based on a simple reduction of constant-time security of a program P to safety of a product program Q that simulates two executions of P. We formalize and verify the reduction for a core high-level language using the Coq proof assistant.

1 Introduction

Timing attacks pose a serious threat to otherwise secure software systems. Such attacks can be mounted by measuring the execution time of an implementation directly in the execution platform [23] or by interacting remotely with the implementation through a network. Notable examples of the latter include Brumley and Boneh's key recovery attacks against OpenSSL's implementation of the RSA decryption operation [15]; and the Canvel et al. [16] and Lucky 13 [4] timing-based padding-oracle attacks, that recover application data from SSL/TLS connections [38]. A different class of timing attacks exploit side-effects of cache-collisions; here the attacker infers memory-access patterns of the target program — which may depend on secret data — from the memory latency correlation created by cache sharing between processes hosted on the same machine [11, 31]. It has been demonstrated in practice that these attacks allow the recovery of secret key material, such as complete AES keys [21].

As a countermeasure, many security practitioners mitigate vulnerability by adopting so-called *constant-time* programming disciplines. A common principle of such disciplines governs programs' control-flow paths in order to protect against attacks based on measuring execution time and branch-prediction attacks, requiring that paths do not depend on program secrets. On its own, this characterization is roughly equivalent to security in the program counter model [29] in which program counter values do not depend on program secrets. Stronger constant-time policies also govern programs' memory-access patterns in order to protect against cache-timing attacks, requiring that accessed memory addresses do not depend on program secrets. Further refinements govern the operands of program operations, e.g., requiring that inputs to certain operations do not depend on program secrets, as the execution time of some machine instructions, notably integer division and floating point operations, may depend on the values of their operands.

Although constant-time security policies are the most effective and widely-used software-based countermeasures against timing attacks [11, 25, 20], writing constant-time implementations can be difficult. Indeed, doing so requires the use of low-level programming languages or compiler knowledge, and forces developers to deviate from conventional programming practices. For instance, the program `if` b `then` $\texttt{x} := v_1$ `else` $\texttt{x} := v_2$ may be replaced with the less conventional $\texttt{x} := b * v_1 + (1-b) * v_2$. Furthermore, the observable properties of a program execution are generally not evident from its source code, e.g., due to optimizations made by compilers or due to platform-specific behaviours.

This raises the question of how to validate constant-time implementations. A recently disclosed timing leak in OpenSSL's DSA signing [19] procedure demonstrates that writing constant-time code is complex and requires some form of validation. The recent case of Amazon's s2n library also demonstrates that the deployment of less rigid timing countermeasures is extremely hard to validate: soon after its release, two patches[1] were issued for protection against timing attacks [3, 5], the second of which exploits a timing-related vulnerability introduced when fixing the first. These vulnerabilities eluded both extensive code review and testing, suggesting that standard software validation processes are an inadequate defense against timing vulnerabilities, and that more rigorous analysis techniques are necessary.

In this work, we develop a unifying formal foundation for constant-time programming policies, along with a formal and fully automated verification technique. Our formalism is parameterized by a flexible leakage model that captures the various constant-time policies used in practice, including path-based, address-based, and operand-based characterizations, wherein program paths, accessed memory addresses, and operand sizes, respectively, are independent of program secrets. Importantly, our formalism is precise with respect to the characterization of program secrets, distinguishing not only between public and private input values, but also between private and publicly observable output values. While this distinction poses technical and theoretical challenges, constant-time implementations in cryptographic libraries like OpenSSL include optimizations for which paths, addresses, and operands are contingent not only on public input values, but also on publicly observable output values. Considering only input values as non-secret information would thus incorrectly characterize those implementations as non-constant-time.

We demonstrate the practicality of our verification technique by developing a prototype, ct-verif, and evaluating it on a comprehensive set of case studies collected from various off-the-shelf libraries such as OpenSSL [25], NaCl [13], FourQlib [17] and curve25519-donna.[2] These examples include a diverse set of constant-time algorithms for fixed-point arithmetic, elliptic curve operations, and symmetric and public-key cryptography. Apart from indicating which inputs and outputs should be considered public, the verification of our examples does not require user intervention, can handle existing (complete and non-modified) implementations, and is fully automated.

One strength of our verification technique is that it is agnostic as to the representation of programs and could be performed on source code, intermediate representations, or machine code. From a theoretical point of view, our approach to verifying constant-time policies is a sound and complete reduction of the security of a program P to the assertion-safety of a program Q, meaning that P is constant-time (w.r.t. the chosen policy) if and only if Q is assertion-safe. We formalize and verify the method for a core high-level language using the Coq proof assistant. Our reduction is inspired from prior work on self-composition [10, 37] and product programs [40, 9], and constructs Q as a product of P with itself—each execution of Q encodes two executions of P. However, our approach is unique in that it exploits the key feature of constant-time policies: program paths must be independent of program secrets. This allows a succinct construction for Q since each path of Q need only correspond to a single control path[3] of P — path divergence of the two executions of P would violate constant-time. Our method is practical precisely because of this optimization: the product program Q has only as many paths as P itself, and its verification can be fully automated.

Making use of this reduction in practice raises the issue of choosing the programming language over which verification is carried out. On the one hand, to obtain a faithful correspondence with the executable program under attacker scrutiny, one wants to be as close as possible to the machine-executed assembly code. On the other hand, building robust and sustainable tools is made easier by existing robust and sustainable frameworks and infrastructure. Our ct-verif prototype performs verification of constant-time properties at the level of optimized LLVM assembly code, which represents a sweet spot in the design space outlined by the above requirements.

Indeed, performing verification after most optimization passes ensures that the program, which may have been written in a higher-level such as C, preserves the constant-time policy even after compiler optimizations. Further, stepping back from machine-specific assembly code to LLVM assembly essentially supports generic reasoning over *all* machine architectures—with the obvious caveat that the leakage model adopted at the LLVM level captures the leakage considered in all the practical lower-level languages and adversary models under study. This is a reasonable assumption, given the small abstraction gap between the two languages. (We further discuss the issues that may arise between LLVM and lower-level assembly

[1]See pull requests #147 and #179 at `github.com/awslabs/s2n`.

[2]`https://code.google.com/p/curve25519-donna/`

[3]This is more subtle for programs with publicly observable outputs; see Section 4.

code when describing our prototype implementation.) Finally, our prototype and case studies justify that existing tools for LLVM are in fact sufficient for our purposes. They may also help inform the development of architecture-specific verification tools.

In summary, this work makes the following fundamental contributions, each described above:

i. a unifying formal foundation for constant-time programming policies used in practice,
ii. a sound and complete reduction-based approach to verifying constant-time programming policies, verified in Coq, and
iii. a publicly available, cross-platform, and fully automated prototype implementing this technique on LLVM code, ct-verif, based on SMACK,
iv. extensive case studies demonstrating the practical effectiveness of our approach on LLVM code, and supporting discussions on the wider applicability of the technique.

We begin in Section 2 by surveying constant-time programming policies. Then in Section 3 we develop a notion of constant-time security parameterized over leakage models, and in Section 4 we describe our reduction from constant-time security to assertion safety on product programs. Section 5 describes our implementation of a verifier for constant-time leveraging this reduction, and in Section 6 we study the verification of actual cryptographic implementations using our method. We discuss related work in Section 7, and conclude in Section 8.

2 Constant-Time Implementations

We now explain the different flavors of constant-time security policies and programming disciplines that enforce them, using small examples of problematic code that arise repeatedly in cryptographic implementations. Consider first the C function of Figure 1, that copies a sub-array of length `sub_len`, starting at index `l_idx`, from array `in` to array `out`. Here, `len` represents the length of array `in`.

```
1 void copy_subarray(uint8 *out, const uint8 *in,
      uint32 len, uint32 l_idx, uint32 sub_len) {
2   uint32 i, j;
3   for(i=0;j=0;i<len;i++) {
4     if (i >= l_idx) && (i < l_idx + sub_len) {
5       out[j] = in[i]; j++;
6     }
7   }
8 }
```

Figure 1: Sub-array copy: `l_idx` is leaked by PC.

Suppose now that the starting addresses and lengths of both arrays are public. What we mean by this is that, the user/caller of this function is willing to accept a contract expressed over the calling interface, whereby the starting addresses and lengths of both arrays may be leaked to an attacker, whereas the value of the `l_idx` variable and the array contents must not. Then, although the overall execution time of this function may seem roughly constant because the loop is executed a number of times that can be inferred from a public input, it might still leak sensitive information via the control flow. Indeed, due to the if condition in line 4, an attacker that is able to obtain a program-counter trace would be able to infer the value of `l_idx`. This could open the way to timing attacks based on execution time measurements, such as the Canvel et al. [16] and Lucky 13 [4] attacks, or to branch-prediction attacks in which a co-located spy process measures the latency introduced by the branch-prediction unit in order to infer the control flow of the target program [1]. An alternative implementation that fixes this problem is shown in Figure 2.

```
1  uint32 ct_lt(uint32 a, uint32 b) {
2    uint32 c = a ^ ((a ^ b) | ((a - b) ^ b));
3    return (0 - (c >> (sizeof(c) * 8 - 1)));
4  }
5
6  void cp_copy_subarray(uint8 *out, const uint8 *in,
         uint32 len, uint32 l_idx, uint32 sub_len) {
7    uint32 i, j, in_range;
8    for(i=0;i<sub_len;i++) out[i]=0;
9    for(i=0,j=0;i<len;i++) {
10     in_range = 0;
11     in_range |= ~ct_lt(i,l_idx);
12     in_range &=  ct_lt(i,l_idx+sub_len);
13     out[j] |= in[i] & in_range;
14     j = j + (in_range % 2);
15   }
16 }
```

Figure 2: Sub-array copy: constant control flow but `l_idx` is leaked by memory access address trace.

Observe that the control flow of this function is now totally independent of `l_idx`, which means that it is constant for fixed values of all public parameters. However, this implementation allows a different type of leakage that could reveal `l_idx` to a stronger class of timing adversaries. Indeed, the memory accesses in line 13 would allow an attacker with access to the full trace of memory addresses accessed by the program to infer the value of `l_idx`—note that the sequence of `j` values will repeat at 0 until `l_idx` is reached, and then increase. This leakage could be exploited via cache-timing attacks [11, 31], in which an attacker controlling a spy process co-located with this program (and hence sharing the same cache) would measure its own memory access times and try to infer sensitive data leaked to accessed addresses from cache hit/miss patterns.

Finally, the program above also includes an additional potential leakage source in line 14. Here, the value of `j` is updated as a result of a DIV operation whose execution time, in some processors,[4] may vary depending on the

[4]This is a quotation from the Intel 64 and IA-32 architectures ref-

values of its operands. This line of code might therefore allow an attacker that can take fine-grained measurements of the execution time to infer the value of `l_idx` [25]. There are two possible solutions for this problem: either ensure that the ranges of operands passed to such instructions are consistently within the same range, or use different algorithms or instructions (potentially less efficient) whose execution time does not depend on their operands. We note that, for this class of timing attackers, identifying leakage sources and defining programming disciplines that guarantee adequate mitigation becomes highly platform-specific.

An implementation of the same function that eliminates all the leakage sources we have identified above—assuming that the used native operations have operand-independent execution times—is given in Figure 3.

```c
uint32 ct_eq(uint32 a, uint32 b) {
  uint32 c = a ^ b;
  uint32 d = ~c & (c - 1);
  return (0 - (d >> (sizeof(d) * 8 - 1)));
}

void ct_copy_subarray(uint8 *out, const uint8 *in,
     uint32 len, uint32 l_idx, uint32 sub_len) {
  uint32 i, j;
  for(i=0;i<sub_len;i++) out[i]=0;
  for(i=0;i<len;i++) {
    for(j=0;j<sub_len;j++) {
      out[j] |= in[i] & ct_eq(l_idx+j,i);
    }
  }
}
```

Figure 3: Constant-time sub-array copy.

It is clear that the trade-off here is one between efficiency and security and, indeed, constant-time implementations often bring with them a performance penalty. It is therefore important to allow for relaxations of the constant-time programming disciplines when these are guaranteed *not* to compromise security. The example of Figure 4, taken from the NaCl cryptographic library [13] illustrates an important class of optimizations that arises from allowing leakage which is known to be *benign*.

This code corresponds to a common sequence of operations in secure communications: first verify that an incoming ciphertext is authentic (line 11) and, if so, recover the enclosed message (line 12) cleaning up some spurious data afterwards (line 13). The typical contract drawn at the function's interface states that the secret inputs to the function include only the contents of the secret key array. Now suppose we ensure that the functions called by this code are constant-time. Even so, this function is *not* constant-time: the result of the verification in line 11 obviously depends on the secret key value, and it is used for a conditional return statement.

```c
 1  int crypto_secretbox_open(unsigned char *m,
 2    const unsigned char *c,unsigned long long clen,
 3    const unsigned char *n,
 4    const unsigned char *k)
 5  {
 6    int i;
 7    unsigned char subkey[32];
 8    if (clen < 32) return -1;
 9    crypto_stream_salsa20(subkey,32,n,k);
10    if (crypto_auth_hmacsha512_verify(c,c+32,clen
        -32,subkey)!=0) return -1;
11    crypto_stream_salsa20_xor(m,c,clen,n,k);
12    for (i = 0;i < 32;++i) m[i] = 0;
13    return 0;
14  }
```

Figure 4: Verify-then-decrypt: verification result is publicly observable and can be leaked by control-flow.

The goal of this return statement is to reduce the execution time by preventing a useless decryption operation when the ciphertext is found to be invalid. Indeed, an authenticated decryption failure is typically publicly signaled by cryptographic protocols, in which case this blatant violation of the constant-time security policy would actually *not* constitute an additional security risk. Put differently, the potentially sensitive bit of information revealed by the conditional return is actually benign leakage: it is safe to leak it because it will be revealed anyway when the return value of the function is later made public. Such optimization opportunities arise whenever the target application accepts a contract at the function interface that is enriched with information about publicly observable outputs, and this information is sufficient to classify the extra leakage as benign.

The above examples motivate the remainder of the work in this paper. It is clear that checking the correct enforcement of constant-time policies is difficult. Indeed, the programming styles that need to be adopted are very particular to this domain, and degrade the readability of the code. Furthermore, these are non-functional properties that standard software development processes are not prepared to address. These facts are usually a source of criticism towards constant-time implementations. However, our results show that such criticism is largely unjustified. Indeed, our verification framework stands as proof that the strictness of constant-time policies makes them suitable for automatic verification. This is not the case for more lenient policies that are less intrusive but offer less protection (e.g., guaranteeing that the total execution time varies within a very small interval, or that the same number of calls is guaranteed to be made to a hash compression function).

erence manual: *The throughput of "DIV/IDIV r32" varies with the number of significant digits in the input EDX:EAX and/or of the quotient of the division for a given size of significant bits in the divisor r32. The throughput decreases (increasing numerical value in cycles) with increasing number of significant bits in the input EDX:EAX or the output quotient. The latency of "DIV/IDIV r32" also varies with the significant bits of the input values. For a given set of input values, the latency is about the same as the throughput in cycles.*

In the next section we formalize constant-time security following the intuition above, as well as the foundations for a new formal verification tool that is able to automatically verify their correct enforcement over real-world cryptographic code.

3 A Formalization of Constant-Time

In order to reason about the security of the code actually executed after compilation, we develop our constant-time theory and verification approach on a generic unstructured assembly language, in Appendix A. In the present section we mirror that development on a simple high-level structured programming language for presentational clarity. We consider the language of *while programs*, enriched with arrays and assert/assume statements. Its syntax is listed in Figure 5. The metavariables x and e range over program variables and expressions, respectively. We leave the syntax of expressions unspecified, though assume they are deterministic, side-effect free, and that array expressions are non-nested.

$$\begin{aligned} p ::= {} & \texttt{skip} \mid x\texttt{[}e_1\texttt{]} \texttt{ := } e_2 \mid \texttt{assert } e \mid \texttt{assume } e \mid p_1 \texttt{; } p_2 \\ & \mid \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2 \mid \texttt{while } e \texttt{ do } p \end{aligned}$$

Figure 5: The syntax of while programs.

Although this language is quite simple, it is sufficient to fully illustrate our theory and verification technique. We include arrays rather than scalar program variables to model constant-time policies which govern indexed memory accesses. We include the assert and assume statements to simplify our reduction from the security of a given program to the assertion-safety of another. Figure 6 lists the semantics of while programs, which is standard.

$$\frac{s' = s[\langle x, s(e_1)\rangle \mapsto s(e_2)]}{\langle s,\ x\texttt{[}e_1\texttt{]} \texttt{ := } e_2\rangle \to \langle s', \texttt{skip}\rangle} \qquad \frac{s' = s \text{ if } s(e) \text{ else } \bot}{\langle s,\ \texttt{assert } e\rangle \to \langle s', \texttt{skip}\rangle}$$

$$\frac{s(e) = \texttt{true}}{\langle s,\ \texttt{assume } e\rangle \to \langle s, \texttt{skip}\rangle} \qquad \frac{\langle s, p_1\rangle \to \langle s', p_1'\rangle}{\langle s,\ p_1\texttt{; } p_2\rangle \to \langle s',\ p_1'\texttt{; } p_2\rangle}$$

$$\frac{}{\langle s,\ \texttt{skip; } p\rangle \to \langle s, p\rangle} \qquad \frac{i = 1 \text{ if } s(e) \text{ else } 2}{\langle s,\ \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2\rangle \to \langle s, p_i\rangle}$$

$$\frac{p' = (p\texttt{; while } e \texttt{ do } p) \text{ if } s(e) \text{ else } \texttt{skip}}{\langle s,\ \texttt{while } e \texttt{ do } p\rangle \to \langle s,\ p'\rangle}$$

Figure 6: The operational semantics of while programs. All rules are guarded implicitly by the predicate $s \neq \bot$, and we abbreviate the predicate $s(e) = \texttt{true}$ by $s(e)$.

A *state* s maps variables x and indices $i \in \mathbb{N}$ to values $s(x,i)$, and we write $s(e)$ to denote the value of expression e in state s. The distinguished *error state* $\bot$ represents a state from which no transition is enabled. A *configuration* $c = \langle s, p\rangle$ is a state s along with a program p to be executed, and an *execution* is a sequence $c_1 c_2 \ldots c_n$ of configurations such that $c_i \to c_{i+1}$ for $0 < i < n$. The execution is *safe* unless $c_n = \langle \bot, \_\rangle$; it is *complete* if $c_n = \langle \_, \texttt{skip}\rangle$; and it is an *execution of program p* if $c_1 = \langle \_, p\rangle$. A program p is *safe* if all of its executions are safe.

A *leakage model* L maps program configurations c to *observations* $L(c)$, and extends to executions, mapping $c_1 c_2 \ldots c_n$ to the *observation* $L(c_1 c_2 \ldots c_n) = L(c_1) \cdot L(c_2) \cdots L(c_n)$, where ε is the identity observation, and $L(c) \cdot \varepsilon = \varepsilon \cdot L(c) = L(c)$. Two executions α and β are *indistinguishable* when $L(\alpha) = L(\beta)$.

Example 1. *The baseline path-based characterization of constant-time is captured by leakage models which expose the valuations of branch conditions:*

$$\langle s,\ \textit{if } e \textit{ then } p_1 \textit{ else } p_2\rangle \mapsto s(e)$$
$$\langle s,\ \textit{while } e \textit{ do } p\rangle \mapsto s(e)$$

In this work we assume that all leakage models include the mappings above.

Example 2. *Notions of constant-time which further include memory access patterns are captured by leakage models which expose addresses accessed in load and store instructions. In our simple language of while programs, this amounts to exposing the indexes to program variables read and written at each statement. For instance, the assignment statement exposes indexes read and written (the base variables need not be leaked as they can be inferred from the control flow):*

$$\langle s,\ x_0\,[e_0]\ \ :=\ e\rangle \mapsto s(e_0)\,s(e_1)\cdots s(e_n)$$

where x_1 [e_1], …, x_n [e_n] are the indexed variable reads in expression e (if any exist).

Example 3. *Notions of constant-time which are sensitive to the size of instruction operands, e.g., the operands of division instructions, are captured by leakage models which expose the relevant leakage:*

$$\langle s,\ x\,[e_1]\ \ :=\ e_2\ /\ e_3\rangle \mapsto S(e_2, e_3)$$

where S is some function over the operands of the division operation, e.g., the maximum size of the two operands.

In Section 2 we have intuitively described the notion of a contract drawn at a function's interface; the constant-time security policies are defined relatively to this contract, which somehow defines the acceptable level of (benign) leakage that can be tolerated. Formally, we capture these contracts using a notion of equivalence between initial states (for a set X_i of inputs declared to be public) and final states (for a set X_o of outputs declared to be publicly observable), as follows.

Given a set X of program variables, two configurations $\langle s_1, \_\rangle$ and $\langle s_2, \_\rangle$ are *X-equivalent* when $s_1(x,i) = s_2(x,i)$ for all $x \in X$ and $i \in \mathbb{N}$. Executions $c_1 \dots c_n$ and $c'_1 \dots c'_{n'}$ are *initially X-equivalent* when c_1 and c'_1 are X-equivalent, and *finally X-equivalent* when c_n and $c'_{n'}$ are X-equivalent.

Definition 1 (Constant-Time Security). *A program is* secure *when all of its initially X_i-equivalent and finally X_o-equivalent executions are indistinguishable.*

Intuitively, constant-time security means that any two executions whose input and output values differ only with respect to secret information must leak exactly the same observations. Contrasting our definition with other information flow policies, we observe that constant-time security asks that every two complete executions starting with X_i-equivalent states and ending with X_o-equivalent final states must be indistinguishable, while termination-insensitive non-interference asks that every two complete executions starting with X_i-equivalent states must end with X_o-equivalent final states. This makes the constant-time policies we consider distinct from the baseline notions of non-interference studied in language-based security. However, our policies can be understood as a specialized form of delimited release [34], whereby *escape hatches* are used to specify an upper bound on the information that is allowed to be declassified. Our notion of security is indeed a restriction of delimited release where escape hatches–our public output annotations–may occur only in the final state.

4 Reducing Security to Safety

The construction of the output-insensitive product of a program (with itself) is shown in Figure 7. It begins by assuming the equality of each public input $x \in X_\mathrm{i}$ with its renamed copy $\hat{x}$, then recursively applies a guard and instrumentation to each subprogram. Guards assert the equality of leakage functions for each subprogram p and its variable-renaming $\hat{p}$.

$\mathsf{product}(p)$	`assume` x=$\hat{x}$ `for` $x \in X_\mathrm{i}$`;` $\mathsf{together}(p)$
$\mathsf{together}(p)$	$\mathsf{guard}(p)$`;` $\mathsf{instrument}[\lambda p.(p\,;\hat{p}), \mathsf{together}](p)$
$\mathsf{guard}(p)$	`assert` $L(p)$=$L(\hat{p})$

Figure 7: Output-insensitive product construction.

Instrumentation preserves the control structure of the original program. Our construction uses the program instrumentation given in Figure 8, which is parameterized by functions α and β transforming assignments and subprograms, respectively. In our constructions, α is either the identity function or else duplicates assignments over renamed variables, and β applies instrumentation recursively with various additional logics.

_	$\mathsf{instrument}[\alpha,\beta](\_)$
`skip`	`skip`
x`[`e_1`] :=` e_2	$\alpha(x$`[`e_1`] :=` $e_2)$
`assert` e	`assert` e
`assume` e	`assume` e
p_1`;` p_2	$\beta(p_1)$`;` $\beta(p_2)$
`if` e `then` p_1 `else` p_2	`if` e `then` $\beta(p_1)$ `else` $\beta(p_2)$
`while` e `do` p	`while` e `do` $\beta(p)$

Figure 8: Instrumentation for product construction.

Our first result states that this construction provides a reduction from constant-time security to safety that is sound for all safe input programs (i.e., a security verdict is always correct) and complete for programs where information about public outputs is not taken into consideration in the security analysis (i.e., an insecurity verdict is always correct).

Theorem 1. *A safe program with (respectively, without) public outputs is secure if (respectively, iff) its output-insensitive product is safe.*

Proof. First, note that program semantics is deterministic, i.e., for any two complete executions of a program p from the same state s_0 to states s_1 and s_2 emitting observations $L(\vec{c}_1)$ and $L(\vec{c}_2)$, respectively, we have $s_1 = s_2$ and $L(\vec{c}_1) = L(\vec{c}_2)$. The product construction dictates that an execution of $\mathsf{product}(p)$ from state $s \uplus \hat{s}$ reaches state $s' \uplus \hat{s}'$ if and only if the two corresponding executions of p leak the same observation sequence, from s to s' and from $\hat{s}$ to $\hat{s}'$, where $\hat{s}$ is the variable-renaming of s. □

In order to deal with program paths which depend on public outputs, we modify the product construction, as shown in Figure 9, to record the observations along output-dependent paths in history variables and assert their equality when paths merge. The output-sensitive product begins and ends by assuming the equality of public inputs and outputs, respectively, with their renamed copies, and finally asserts that the observations made across both simulated executions are identical. Besides delaying the assertion of observational indistinguishability until the end of execution, when outputs are known to be publicly observable, this construction allows paths to diverge at branches which depend on these outputs, checking whether both executions alone leak the same recorded observations.

Technically, this construction therefore relies on identifying the branches, i.e., the if and while statements, whose conditions can only be declared benign when public outputs are considered. This has two key implications.

```
product($p$)       same_observations := true;
                   assume $x$=$\hat{x}$ for $x \in X_i$;
                   together($p$);
                   assume $x$=$\hat{x}$ for $x \in X_o$;
                   assert same_observations

together($p$)      if benign($p$) then
                     h := $\varepsilon$; $\hat{h}$ := $\varepsilon$;
                     alone$_h$($p$);
                     alone$_{\hat{h}}$($\hat{p}$);
                     same_observations &&:= h=$\hat{h}$
                   otherwise
                     guard($p$);
                     instrument[$\lambda p.(p;\hat{p})$,together]($p$)

guard($p$)         same_observations &&:= $L(p)$=$L(\hat{p})$

alone$_h$($p$)     record$_h$($p$);
                   instrument[$\lambda p.p$,alone$_h$]($p$)

record$_h$($p$)    $h$ +:= $L(p)$
```

Figure 9: Output-sensitive product construction

First, it either requires a programmer to annotate which branches are benign in a public-sensitive sense, or additional automation in the verifier, e.g., to search over the space of possible annotations; in practice the burden appears quite low since very few branches will need to be annotated as being benign. Second, it requires the verifier to consider separate paths for the two simulated executions, rather than a single synchronized path. While this deteriorates to an expensive full product construction in the worst case, in practice these output-dependent branches are localized to small non-nested regions, and thus asymptotically insignificant.

Theorem 2. *A safe program is secure iff its output-sensitive product is safe with some benign-leakage annotation.*

Proof. Completeness follows from completeness of self-composition, so only soundness is interesting. Soundness follows from the fact that we record history in the variables h and $\hat{h}$ whenever we do not assert the equality of observations on both sides. □

Coq formalization The formal framework presented in this and the previous section has been formalized in Coq. Our formalization currently includes the output-insensitive reduction from constant-time security to safety of the product program as described in Figures 7 and 8, for the while language in Figure 5. We prove the soundness and completeness of this reduction (Theorem 1) following the intuition described in the sketch presented above. Formalization of the output-sensitive construction and the proof of Theorem 2 should not present any additional difficulty, other than a more intricate case analysis when control flow may diverge. Our Coq formalization serves two purposes: i. it rigorously captures the theoretical foundations of our approach and complements the intuitive description we gave above; and ii. it could serve as a template for a future formalization of the machine-level version of these same results, which underlies the implementation of our prototype and is presented in Appendices A and B. A Coq formalization of this low-level transformation could be integrated with CompCert, providing more formal guarantees on the final compiled code.

5 Implementation of a Security Verifier

Using the reduction of Section 4 we have implemented a prototype, ct-verif, which is capable of automatically verifying the compiled and optimized LLVM code resulting from compiling actual C-code implementations of several constant-time algorithms. Before discussing the verification of these codes in Section 6, here we describe our implementation and outline key issues. Our implementation and case studies are publicly available[5] and cross-platform. ct-verif leverages the SMACK verification tool [32] to compile the annotated C source via Clang[6] and to optimize the generated assembly code via LLVM[7] before translating to Boogie[8] code. We perform our reduction on the Boogie code, and apply the Boogie verifier (which performs verification using an underlying SMT[9] logic solver) to the resulting program.

5.1 Security Annotations

We provide a simple annotation interface via the following C function declarations:

```
void public_in(smack_value_t);
void public_out(smack_value_t);
void benign_branching();
```

where `smack_value_t` values are handles to program values obtained according to the following interface

```
smack_value_t __SMACK_value();
smack_value_t __SMACK_values(void* ary,
                             unsigned count);
smack_value_t __SMACK_return_value(void);
```

and `__SMACK_value(x)` returns a handle to the value stored in program variable `x`, `__SMACK_values(ary,n)` returns a handle to an `n`-length array `ary`, and `__SMACK_return_value()` provides a handle to the procedure's return value. While our current interface does not provide handles to entire structures, non-recursive structures can still be annotated by annotating the handles to each of their (nested) fields. Figure 10 demonstrates the annotation of a decryption function for the Tiny Encryption Algorithm (TEA). The first argument `v` is a pointer to

[5] https://github.com/imdea-software/verifying-constant-time
[6] C language family frontend for LLVM: http://clang.llvm.org
[7] The LLVM Compiler Infrastructure: http://llvm.org
[8] Boogie: http://github.com/boogie-org/boogie
[9] Satisfiability Modulo Theories: http://smtlib.cs.uiowa.edu

a public ciphertext block of two 32-bit words, while the second argument k is a pointer to a secret key.

```
void decrypt_cpa_wrapper(uint32_t* v,uint32_t* k){
  public_in(__SMACK_value(v));
  public_in(__SMACK_value(k));
  public_in(__SMACK_values(v, 2));
  decrypt(v, k);
}
```

Figure 10: Annotations for the TEA decryption function.

5.2 Reasoning about Memory Separation

In some cases, verification relies on establishing separation of memory objects. For instance, if the first of two adjacent objects in memory is annotated as public input, while the second is not, then a program whose branch conditions rely on memory accesses from the first object is only secure if we know that those accesses stay within the bounds of the first object. Otherwise, if those accesses might occur within the second object, then the program is insecure since the branch conditions may rely on private information.

Luckily SMACK has builtin support for reasoning about the separation of memory objects, internally leveraging an LLVM-level data-structure analysis [26] (DSA) to partition memory objects into disjoint regions. Accordingly, the generated Boogie code encodes memory as several disjoint map-type global variables rather than a single monolithic map-type global variable, which facilitates scalable verification. This usually provides sufficient separation for verifying security as well. In a few cases, DSA may lack sufficient precision. In those settings, it would be possible to annotate the source code with additional assumptions using SMACK 's `__VERIFIER_assume()` function. This limitation is not fundamental to our approach, but instead an artifact of design choices[10] and code rot[11] in DSA itself.

5.3 Product Construction for Boogie Code

The Boogie intermediate verification language (IVL) is a simple imperative language with well-defined, clean, and mathematically-focused semantics which is a convenient representation for performing our reduction. Conceptually there is little difference between performing our shadow product reduction at the Boogie level as opposed to the LLVM or machine-code level since the Boogie code produced by SMACK corresponds closely to the LLVM code, which is itself similar to machine code. Indeed our machine model of Appendix A is representative. Practically however, Boogie's minimal syntax greatly facilitates our code-to-code translation. In particular, shadowing the machine state amounts to making duplicate copies of program variables. Since memory accesses are represented by accesses to map-type global variables, accessing a shadowed address space amounts to accessing the duplicate of a given map-type variable.

Our prototype models observations as in Examples 1 and 2 of Section 3, exposing the addresses used in memory accesses and the values used as branch conditions as observations. According to our construction of Section 4, we thus prefix each memory access by an assertion that the address and its shadow are equal, and prefix each branch by an assertion that the condition and its shadow are equal. Finally, for procedures with annotations, our prototype inserts assume statements on the equality of public inputs with their shadows at entry blocks.

When dealing with public outputs, we perform the output-sensitive product construction described in Section 4 adapted to an unstructured assembly language. Intuitively, our prototype delays assertions (simply by keeping track of their conjunction in a special variable) but otherwise produces the standard output-insensitive product program. It then replaces the blocks corresponding to the potentially desynchronized conditional with blocks corresponding to the output-sensitive product construction that mixes control and data product. Finally, it inserts code that saves the current assertions before the region where the control flow may diverge, and restores them afterwards, making sure to also take into account the assertions collected in between.

5.4 Scalability of the Boogie Verifier

Since secure implementations, and cryptographic primitives in particular, do not typically call recursive procedures, we instruct Boogie to inline all procedures during verification. This avoids the need for manually written procedure contracts, or for sophisticated procedure specification inference tools.

Program loops are handled by automatically computing loop invariants. This is fairly direct in our setting, since invariants are simply conjunctions of equalities between some program variables and their shadowed copies. We compute the relevant set of variables by taking the intersection of variables live at loop heads with those on which assertions inserted by our reduction depend.

5.5 Discussion

ct-verif is based on a theoretically sound and complete methodology; however, practical interpretations of its results must be analyzed with care. First, leakage models are constructed, and in our case are based on LLVM rather than machine code. Second, verification tools can

[10] DSA is designed to be extremely scalable at the expense of precision, yet such extreme scalability is not necessary for our use.

[11] See the discussion thread at `https://groups.google.com/forum/#!topic/llvm-dev/pnU5ecuvr6c`.

be incomplete, notably because of approximations made by sub-tasks performed during verification (for instance, data-structure analysis or invariant inference).

Therefore, it is important to evaluate ct-verif empirically, both on positive and negative examples. Our positive experimental results in the next section demonstrate that the class of constant-time programs that is validated automatically by ct-verif is significantly larger than those tackled by existing techniques and tools. Our negative examples, available from the public repository,[12] are taken from known buggy libraries (capturing the recent CacheBleed attack,[13] in particular), and others taken to illustrate particularly tricky patterns. Again, some of these examples illustrate the value of a low-level verification tool by exhibiting compilation-related behaviours. Unsurprisingly, we found that there is little value in running our tool on code that was *not* written to be constant-time. Conversely, we found that our tool can be helpful in detecting subtle breaches in code that was written with constant-time in mind, but was still insecure, either due to subtle programming errors, or to compilation-related issues.

It remains to discuss possible sources of unsoundness that may arise from our choice of LLVM as the target for verification (rather than actual machine code). As highlighted in Section 1, this choice brings us many advantages, but it implies that our prototype does not strictly know what machine instructions will be activated and on which arguments, when the final code is actually executed. For example, our assumptions on the timing of a particular LLVM operation may not hold for the actual processor instruction that is selected to implement this operation in executable code. Nevertheless we argue that the LLVM assembly code produced just before code generation sufficiently similar to *any* target-machine's assembly code to provide a high level of confidence. Indeed, the majority of compiler optimizations are made prior to code generation. At the point of code generation, the key difference between representations is that in LLVM assembly:

i. some instruction/operand types may not be available on a given target machine,
ii. there are arbitrarily-many registers, whereas any given machine would have a limited number, and
iii. the order of instructions within basic blocks is only partially determined.

First we note that neither of these differences affects programs' control-flow paths, and the basic-block structure of programs during code generation is generally preserved. Second, while register allocation does generally change memory-access patterns, spilled memory accesses are generally limited to the addresses of scalar stack variables, which are fully determined by control-flow paths. Thus both path-based and address-based constant-time properties are generally preserved. Operand-based constant-time properties, however, are generally not preserved: it is quite possible that instruction selection changes the types of some instruction's operands, implying a gap between LLVM and machine assembly regarding whether operand sizes may depend on secrets. Dealing with such sources of leakage requires architecture-specific modeling and tools, which are out of the scope of a research prototype.

6 Experimental Results

We evaluate ct-verif on a large set of examples, mostly taken from off-the-shelf cryptographic libraries, including the pervasively used OpenSSL [25] library, the NaCl [13] library, the FourQlib library [17], and `curve25519-donna`.[14] The variety and number of cryptographic examples we have considered is unprecedented in the literature. Furthermore, our examples serve to demonstrate that ct-verif outperforms previous solutions in terms of scale (the sizes of some of our examples are orders of magnitude larger than what could be handled before), coverage (we can handle top-level public APIs, rather than low-level or leaf functions) and robustness (ct-verif is based on a technique which is not only sound, but also complete).

All execution times reported in this section were obtained on a 2.7GHz Intel i7 with 16GB of RAM. Size statistics measure the size in lines of code (loc) of the analyzed Boogie code (similar in size to the analyzed LLVM bitcode) before inlining. When presenting time measurements, all in seconds, we separate the time taken to produce the product program (annotating it with the $\times$ symbol) from that taken to verify it: in particular, given a library, the product program can be constructed once and for all before verifying each of its entry points. ct-verif assumes that the leakage trace produced by standard library functions `memcpy` and `memset` depends only on their arguments (that is, the address and length of the objects they work on, rather than their contents). This is a mild assumption that can be easily checked for each platform. For examples that use dynamic memory allocation, such as the OpenSSL implementation of PKCS#1 padding, ct-verif enforces that `malloc` and `free` are called with secret-independent parameters and assumes that the result of `malloc` is always secret-independent in this case. In other words, we assume that the address returned by `malloc` depends only on the trace of calls to `malloc` and `free`, or that the memory allocator observes only the memory layout to make allocation decisions.[15]

[12] `https://github.com/imdea-software/verifying-constant-time`

[13] `https://ssrg.nicta.com.au/projects/TS/cachebleed`

[14] `https://code.google.com/p/curve25519-donna/`

[15] It may be possible to extend this to an allocator that also has access to the trace of memory accesses, since they are made public.

Example	Size	Time (×)	Time
tea	200	2.33	0.47
rlwe_sample	400	5.78	0.65
nacl_salsa20	700	5.60	1.11
nacl_chacha20	10000	8.30	1.92
nacl_sha256_block	20000	27.7	4.17
nacl_sha512_block	20000	39.49	4.29

Table 1: Verification of crypto primitives.

6.1 Cryptographic Primitives

For our first set of examples, we consider a representative set of cryptographic primitives: a standard implementation of TEA [39] (`tea`), an implementation of sampling in a discrete Gaussian distribution by Bos et al. [14] (`rlwe_sample`) and several parts of the NaCl library [13] library.

Table 1 gives the details (we include only a subset of the NaCl verification results listed as `nacl_xxxx`). The verification result for `rlwe_sample` only excludes its core random byte generator, essentially proving that this core primitive is the only possible source of leakage. In particular, if its leakage and output bytes are independent, then the implementation of the sampling operation is constant-time. Verification of the SHA-256 implementation in NaCl above refers to the compression funcion; the full implementation of the hash function, which simply iterates this compression function, poses a challenge to our prototype due to the action of DSA: the internal state of the function is initialized as a single memory block, that later stores both secret and public values that are accessed separately. This issue was discussed in Section 5.2, where we outlined a solution using assume statements.

6.2 TLS Record Layer

To further illustrate scalability to large code bases, we now consider problems related to the MAC-then-Encode-then-CBC-Encrypt (MEE-CBC) construction used in the TLS record layer to obtain an authenticated encryption scheme. This construction is well-understood from the perspective of provable security [24, 30], but implementations have been the source of several practical attacks on TLS via timing side-channels [16, 4].

We apply our prototype to two C implementations of the MEE-CBC decryption procedure, treating only the input ciphertext as public information. Table 2 shows the corresponding verification results. We extract the first implementation from the OpenSSL sources (version 0.9.8zg). It includes all the countermeasures against timing attacks currently implemented in the MEE-CBC component in OpenSSL as documented in [25]. We verify the parts of the code that handle MEE-CBC decryption (1K loc of C, or 10K loc in Boogie): i. decryption of the encrypted message using AES128 in CBC mode; ii. removing the padding and checking its well-formedness; iii. computing the HMAC of the unpadded message, even for bad padding, and using the same number of calls to the underlying hash compression function (in this case SHA-1); and iv. comparing the transmitted MAC to the computed MAC in constant-time. Our verification does not include the SHA1 compression function and AES-128 encryption—these are implemened in assembly—and hence our result proves that the only possible leakage comes from these leaf functions. (In OpenSSL the SHA1 implementation is constant-time but AES-128 makes secret-dependent memory accesses.)

Example	Time (×)	Time
mee-cbc-openssl	10.6	18.73
mee-cbc-nacl	24.64	92.56

Table 2: Verification of MEE-CBC TLS record layer.

As our second example, we consider a full 800 loc (in C, 20K loc in Boogie) implementation of MEE-CBC [5], which includes the implementation of the low level primitives (taken from the NaCl library).

Our prototype is able to verify the constant-time property of both implementations–with only the initial ciphertext and memory layout marked as public. Perhaps surprisingly, our simple heuristic for loop invariants is sufficient to handle complex control-flow structures such as the one shown in Figure 11, taken from OpenSSL.

```
k = 0;
if (/* low cond */) { k = /* low exp */ }
if (k > 0)
{ for (i = 1; i < k / /* low var */; i++)
  { /* i-dependent memory access */ }
}
for (i = /* low var */; i <= /* low var */; i++)
{ /* i-dependent memory access */
  for (j = 0; j < /* low var */; j++)
  { if (k < /* low var */)
      /* k-dependent memory access */
    else if (k < /* low exp */)
      /* k-dependent memory access */
    k++;
  }
  for (j = 0; j < /* low var */; j++)
  { /* j-dependent memory access */ }
}
```

Figure 11: Complex control-flow from OpenSSL.

6.3 Fixed-Point Arithmetic

Our third set of examples is taken from the `libfixedtimefixedpoint` library, developed by Andrysco et al. [7] to mitigate several attacks due to operand-dependent leakage in the timing of floating point operations. In the conclusion of the paper we discuss how our prototype can be extended to deal with the vulnerable code that was attacked in [7]. Here we present

Function	Size	Time
`fix_eq`	100	1.45
`fix_cmp`	500	1.44
`fix_mul`	2300	1.50
`fix_div`	1000	1.53
`fix_ln`	11500	2.66
`fix_convert_from_int64`	100	1.43
`fix_sin`	800	1.64
`fix_exp`	2200	1.62
`fix_sqrt`	1400	1.55
`fix_pow`	18000	1.42[16]

Table 3: Verification of `libfixedtimefixedpoint`.

Example	Size	Time (×)	Time
`curve25519-donna`	10000	10.18	456.97
FourQLib	-	7.87	-
`eccmadd`	2500	-	133.72
`eccdouble`	3000	-	70.67
`eccnorm`	3500	-	156.48
`point_setup`	600	-	0.99
`R1_to_R2`	2500	-	7.92
`R5_to_R1`	2000	-	1.26
`R1_to_R3`	2500	-	2.42
`R2_to_R4`	1000	-	0.93

Table 4: Verification of elliptic curve arithmetic.

our verification results over the library that provides an alternative secure constant-time solution

The `libfixedtimefixedpoint` library (ca. 4K loc of C or 40K loc in Boogie) implements a large number of fixed-point arithmetic operations, from comparisons and equality tests to exponentials and trigonometric functions. Core primitives are automatically generated parametrically in the size of the integer part: we verify code generated with the default parameters. As far as we know, this is the first application of verification to this floating point library.

Table 3 shows verification statistics for part of the library. We verify all arithmetic functions without any inputs marked as public, but display only some interesting data points here. We discuss the `fix_pow` function, during whose execution the code shown in Figure 12 is executed on a `frac` array that is initialized as a `‘‘0’’` string literal. The function in which this snippet appears is not generally constant-time, but it is always used in contexts where all indices of `frac` that are visited up to and including the particular index that might trigger a sudden loop exit at line 6 contains public (or constant) data. Thanks to our semantic characterization of constant-time policies, ct-verif successfully identifies that the leakage produced by this code is indeed benign, whereas existing type-based or taint-propagation based would mark this program as insecure.

```
1 uint64_t result = 0;
2 uint64_t extra  = 0;
3
4 for(int i = 0; i < 20; i++) {
5   uint8_t digit = (frac[i] - (uint8_t) '0');
6   if (frac[i] == '\0') { break; }
7   result+= ((uint64_t)digit) * pow10[i];
8   extra += ((uint64_t)digit) * pow10_extra[i];
9 }
```

Figure 12: `fix_pow` code.

[16]We manually provide an invariant of the form $\exists i_{\mathsf{max}}.\ 0 \le i < i_{\mathsf{max}} \le 20 \wedge \mathtt{frac}[i_{\mathsf{max}}] == 0 \wedge \forall j.\ 0 \le j \le i_{\mathsf{max}} \Rightarrow \mathsf{public}(\mathtt{frac}[j])$ for the loop shown in Figure 12. Loop unrolling could also be used, since the loop is statically bounded.

6.4 Elliptic Curve Arithmetic

As a final illustrative example of the capabilities of ct-verif in handling existing source code from different sources, we consider two constant-time implementations of elliptic curve arithmetic: the `curve25519-donna` implementation by Langley,[17] and the FourQlib library [17]. The former library provides functions for computing essential elliptic curve operations over the increasingly popular Curve25519 initially proposed by Bernstein [12], whereas the latter uses a recently proposed alternative high-performance curve. Table 4 shows the results.

For `curve25519-donna`, we verify the functional entry point, used to generate public points and shared secrets, assuming only that the initial memory layout is public.

For FourQLib, we verify all the core functions for point addition, doubling and normalization, as well as coordinate conversions, all under the only assumption that the addresses of the function parameters are public. ct-verif successfully detects expected dependencies of the execution time on public inputs in the point validation function `ecc_point_validate`.

6.5 Publicly Observable Outputs

We wrap up this experimental section by illustrating the flexibility of the output-sensitive product construction, and how it permits expanding the coverage of real-world crypto implementations in comparison with previous approaches. As a first example we consider an optimized version of the `mee-cbc-nacl` example. Instead of using a constant-time select and zeroing loop to return its result (as shown in Figure 13, where the return code `res` is secret-dependent and marked as public and `in_len` is a public input), the code branches on the return code as shown in Figure 14. (The rest of the code is unmodified, and therefore constant-time.)

This is similar to the motivating example that we presented in Section 2, but here the goal is to avoid the unnecessary cleanup loop at the end of the function in executions where it is not needed. Again, because the return code is made public when it is returned to the caller,

[17]`https://code.google.com/p/curve25519-donna/`

```
good = ~((res == RC_SUCCESS) - 1);
for(i = 0;i < in_len;i++) { out[i] &= good; }
*out_len &= good;
```

Figure 13: MEE-CBC decryption: constant-time.

```
if (res != RC_SUCCESS) {
  for(i = 0;i < in_len;i++) { out[i] = 0; }
  *out_len = 0;
}
```

Figure 14: MEE-CBC decryption: constant-time.

this control-flow dependency on secret information can be classified as benign leakage. The output-sensitive product constructed by our prototype for this example, when the displayed conditional is annotated as benign leakage, verifies in slightly less than 2 minutes. The additional computation cost of verifying this version of the program may be acceptable when compared to the performance gains in the program itself—however minor: verification costs are one off, whereas performance issues in the cryptographic library are paid per execution.

```
 1  int RSA_padding_check_PKCS1_type_2(uchar *to, int
      tlen, const uchar *from, int flen, int num)
 2  {
 3    int i, zero_index = 0, msg_index, mlen = -1;
 4    uchar *em = NULL;
 5    uint good, found_zero_byte;
 6
 7    if (tlen < 0 || flen < 0) return -1;
 8    if (flen > num) goto err;
 9    if (num < 11) goto err;
10
11    em = OPENSSL_zalloc(num);
12    if (em == NULL) return -1;
13    memcpy(em + num - flen, from, flen);
14
15    good = ct_is_zero(em[0]);
16    good &= ct_eq(em[1], 2);
17
18    found_zero_byte = 0;
19    for (i = 2; i < num; i++) {
20      uint equals0 = ct_is_zero(em[i]);
21      zero_index = ct_select_int(~found_zero_byte &
         equals0, i, zero_index);
22      found_zero_byte |= equals0;
23    }
24
25    good &= ct_ge((uint)(zero_index), 2 + 8);
26    msg_index = zero_index + 1;
27    mlen = num - msg_index;
28    good &= ct_ge((uint)(tlen), (uint)(mlen));
29
30    /* We can't continue in constant-time because we
          need to copy the result and we cannot fake
         its length. This unavoidably leaks timing
         information at the API boundary. */
31    if (!good) { mlen = -1; goto err; }
32    memcpy(to, em + msg_index, mlen);
33
34   err:
35    OPENSSL_free(em);
36    return mlen;
37  }
```

Figure 15: RSA PKCS1 padding check from OpenSSL

Finally, we present in Figure 15 an RSA PKCS1.5 padding check routine extracted from OpenSSL (similar code exists in other cryptographic libraries, such as boringssl[18]). The developers note the most interesting feature of this code in the comment on line 30: although this function is written in the constant-time style, the higher-level application (here referred to as an API boundary) does not give this implementation enough information to continue without branching on data dependent from secret inputs (here, the contents of `from`). One way in which this could be achieved would be for the function to accept an additional argument indicating some public bound on the expected message length. The constant-time techniques described previously in this paper could then be used to ensure that the leakage depends only on this additional public parameter. However, given the constraint forced upon the implementer by the existing API, the final statements in the function must be as they are, leading to (unavoidable and hence) benign leakage. Using our techniques, this choice can be justified by declaring the message length returned by the function as being (the only) public output that is safe to leak. Note that `flen`, `tlen` and `num` are public, and hence declaring `mlen` as a public output provides sufficient information to verify the control-flow leakage in line 31, and also the accessed addresses in line 32 as being benign. Verifying the output-sensitive product program when `mlen` is marked as a public output takes under a second.[19]

This example shows that dealing with relaxations of the constant-time policies enabled by output-sensitive API contracts is important when considering functions that are directly accessible by the adversary, rather than internal functions meant to be wrapped. Dealing with these use cases is an important asset of our approach, and is a problem not considered by previous solutions.

7 Related Work

Product programs Product constructions are a standard tool in the algorithmic analysis of systems. Product programs can be viewed as their syntactic counterpart. Systematic approaches for defining and building product programs are considered in [9]. Three instances of product programs are most relevant to our approach: self-composition [10] and selective self-composition [37], which have been used for proving information flow security of programs, and cross-products [40], which have been used for validating the correctness of compiler optimizations. We review the three constructions below, obliviating their original purpose, and presenting them from the perspective of our work.

[18]`https://boringssl.googlesource.com/`

[19]With simple implementations of `OPENSSL_zalloc` and `OPENSSL_free` that wrap standard memory functions.

The self-composition of a program P is a program Q which executes P twice, sequentially, over disjoint copies of the state of P. Self-composition is *sound and complete*, in the sense that the set of executions of program Q is in 1-1 bijection with the set of pairs of executions of program P. However, reasoning about self-composed programs may be very difficult, as one must be able to correlate the actions of two parts of program Q which correspond to the same point of execution of the original program P.

The cross-product Q of a program P coerces the two copies of P to execute in lockstep, by inserting assert statements at each branching instruction. Cross-product is not complete for all programs, because pairs of executions of program P whose control-flow diverge result in an unsafe execution of program Q. As a consequence, one must prove that Q is safe in order to transfer verification results from Q to P. However, cross-product has a major advantage over self-composition: reasoning about cross-products is generally easier, because the two parts of program Q which correspond to the same point of execution of the original program P are adjacent.

Selective self-composition is an approach which alternates between cross-product and self-composition, according to user-provided (or inferred for some applications) annotations. Selective self-composition retains the soundness and completeness of self-composition whilst achieving the practicality of cross-product.

Our output-insensitive product construction (Figure 7) is closely related to cross-product. In particular, Theorem 1 implies that cross-products characterize constant-path programs. We emphasize that, for this purpose, the incompleteness of cross-products is not a limitation but a desirable property. On the other hand, our output-sensitive product construction (Figure 9) is closely related to selective self-composition.

Language-based analysis/mitigation of side-channels Figure 16 summarizes the main characteristics of several tools for verifying constant-time security, according to the level at which they carry out the analysis, the technique they use, their support for public inputs and outputs, their soundness and completeness, and their usability. ct-verif is the only one to support publicly observable outputs, and the only one to be sound, theoretically complete and practical. Moreover, we argue that extending any of these tools to publicly observable outputs is hard; in particular, several of these tools exploit the fact that cryptographic programs exhibit "abnormally straight line code behavior", and publicly observable outputs are precisely used to relax this behavior. We elaborate on these points below.

FlowTracker [33] implements a precise, flow sensitive, constant-time (static) analysis for LLVM programs. This tool takes as input C or C++ programs with security annotations and returns a positive answer or a counterexample.

Tool	Target	Analysis method	Inputs/ Outputs	Sound/ Complete	Usability
tis-ct	C	static	✗/✗	✓/✗	✓(a)
ABPV [6]	C	logical	✓/✗	✓/✓	✗(b)
VirtualCert	x86	static	✓/✗	✓/✗	✗(c)
FlowTracker	LLVM	static	✓/✗	✓/✗	✓
ctgrind	binary	dynamic	✓/✗	✗/✗	✓
CacheAudit	binary	static	✗/✗	✓/✗	✗(d)
This work	LLVM	logical	✓/✓	✓/✓	✓

Figure 16: Comparison of different tools. Target indicates the level at which the analysis is performed. Input/Outputs classifies whether the tool supports public inputs and publicly observable outputs. Usability includes coverage and automation. (a): requires manual interpretation of dependency analysis. (b): requires interactive proofs. (c): requires code rewriting. (d): supports restricted instruction set.

FlowTracker is incomplete (i.e. rejects secure programs), and it does not consider publicly observable outputs.

VirtualCert [8] instruments the CompCert certified compiler [27] with a formally verified, flow insensitive type system for constant-time security. It takes as input a C program with security annotations and compiles it to (an abstraction of) x86 assembly, on which the analysis is performed. VirtualCert imposes a number of restrictions on input programs (so off-the-shelf programs must often be adapted before analysis), is incomplete, and does not support publicly observable outputs.

ctgrind[20] is an extension of Valgrind that verifies constant-address security. It takes an input a program with taint annotations and returns a yes or no answer. ctgrind is neither sound nor complete and does not support publicly obervable outputs.

tis-ct is an extension of the FramaC platform for analyzing dependencies in C programs and helping towards proving constant-time security.[21] tis-ct has been used to analyze OpenSSL. Rather than a verification result, tis-ct outputs a list of all input regions that may flow into the leakage trace, as well as the code locations where that flow may occur. Although this does not directly allow the verification of adherence to a particular security policy, we note that checking that the result list is a subset of public inputs could provide, given an appropriate annotation language, a verification method for public input policies. Since it relies on a dependency analysis rather than semantic criteria, tis-ct is incomplete.

Almeida, Barbosa, Pinto and Vieira [6] propose a methodology based on deductive verification and self-composition for verifying constant-address security of C

[20] `https://github.com/agl/ctgrind/`.

[21] `http://trust-in-soft.com/tis-ct/`

implementations. Their approach extends to constant-address security earlier work by Svenningsson and Sands [36] for constant-path security. This approach does not consider publicly observable outputs and it does not offer a comparable degree of automation to the one we demonstrate in this paper.

CacheAudit [18] is a static analyzer for quantifying cache leakage in a *single run* of a binary program. CacheAudit takes as input a binary program (in a limited subset of 32-bit x86, e.g. no dynamic jump) and a leakage model, but *no* security annotation (there is no notion of public or private, neither for input, nor output). CacheAudit is sound with respect to a simplified machine code semantics, rather than to a security policy. However, it is incomplete.

There are many other works that develop language-based methods for side-channel security (not necessarily in the computational model of this paper). Agat [2] proposes a type-based analysis for detecting timing leaks and a type-directed transformation for closing leaks in an important class of programs. Molnar, Piotrowski, Schultz and Wagner [29] define the program counter security model and a program transformation for making programs secure in this model. Other works include [28, 35, 41].

8 Conclusion

This paper leaves interesting directions for future work. We intend to improve ct-verif in two directions. First, we shall enhance enforcement of more expressive policies, such as those taking into consideration input-dependent instruction execution times. The work of Andrysco et al. [7] shows that variations in the timing of floating point processor operations may lead to serious vulnerabilities in non-cryptographic systems. Dealing with such timing leaks requires reasoning in depth about the semantics of a program, and is beyond the reach of techniques typically used for non-interference analysis. Our theoretical framework inherits this ability from self-composition, and this extension of ct-verif hinges solely on the effort required to embed platform-specific policy specifications into the program instrumentation logics of the prototype. As a second improvement to ct-verif, we will add support for SSE instructions, which are key to reconciling high-speed and security, for example in implementing AES [22].

Acknowledgements The first two authors were funded by Project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020", which is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). The third and fourth authors were supported by projects S2013/ICE-2731 N-GREENS Software-CM and ONR Grants N000141210914 (AutoCrypt) and N000141512750 (SynCrypt). The fourth author was also supported by FP7 Marie Cure Actions-COFUND 291803 (Amarout II).

We thank Peter Schwabe for providing us with a collection of negative examples. We thank Hovav Shacham, Craig Costello and Patrick Longa for helpful observations on our verification results.

References

[1] Onur Aciicmez, Cetin Kaya Koc, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *2007 ACM Symposium on Information, Computer and Communications security (ASIACCS'07)*, pages 312–320. ACM Press, 2007.

[2] Johan Agat. Transforming out Timing Leaks. In *Proceedings POPL'00*, pages 40–53. ACM, 2000.

[3] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on Amazon's s2n implementation of TLS. Cryptology ePrint Archive, Report 2015/1129, 2015. Available at `http://eprint.iacr.org/`. To appear in proceedings of EuroCrypt, 2016.

[4] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy, SP 2013*, pages 526–540. IEEE Computer Society, 2013.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and Francois Dupressoir. Verifiable side-channel security of cryptographic implementations: constant-time mee-cbc. Cryptology ePrint Archive, Report 2015/1241, 2015. Available at `http://eprint.iacr.org/`. To appear in proceedings of Fast Software Encryption, 2016.

[6] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.*, 78(7):796–812, 2013.

[7] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 623–639. IEEE Computer Society, 2015.

[8] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 1267–1279. ACM Press, November 2014.

[9] Gilles Barthe, Juan Manuel Crespo, and Cesar Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *Formal Methods*, volume 6664 of *LNCS*. Springer-Verlag, 2011.

[10] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Computer Security Foundations*, pages 100–114. IEEE Press, 2004.

[11] Daniel J. Bernstein. Cache-timing attacks on aes, 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[12] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006.

[13] Daniel J. Bernstein. Cryptography in NaCl, 2011. `http://nacl.cr.yp.to`.

[14] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 553–570. IEEE Computer Society, 2015.

[15] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[16] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 583–599. Springer, Heidelberg, August 2003.

[17] Craig Costello and Patrick Longa. Four**Q**: Four-dimensional decompositions on a **Q**-curve over the mersenne prime. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 214–235. Springer, 2015.

[18] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Usenix Security 2013*, 2013.

[19] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time". Cryptology ePrint Archive, Report 2016/594, 2016.

[20] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "make sure dsa signing exponentiations really are constant-time". Cryptology ePrint Archive, Report 2016/594, 2016. `http://eprint.iacr.org/2016/594`.

[21] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505. IEEE Computer Society, 2011.

[22] Mike Hamburg. Accelerating AES with vector permute instructions. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 18–32, 2009.

[23] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, Heidelberg, August 1996.

[24] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 310–331. Springer, Heidelberg, August 2001.

[25] Adam Langley. Lucky thirteen attack on TLS CBC. Imperial Violet, February 2013. `https://www.imperialviolet.org/2013/02/04/luckythirteen.html`, Accessed October 25th, 2015.

[26] Chris Lattner, Andrew Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 278–289. ACM, 2007.

[27] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 42–54. ACM, 2006.

[28] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *CSF 2013*, pages 51–65, 2013.

[29] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Dongho Won and Seungjoo Kim, editors, *ICISC 05*, volume 3935 of *LNCS*, pages 156–168. Springer, Heidelberg, December 2006.

[30] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 372–389. Springer, Heidelberg, December 2011.

[31] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[32] Zvonimir Rakamaric and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014.

[33] Bruno Rodrigues, Fernando Pereira, and Diego Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of Compiler Construction*, 2016. To appear.

[34] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, pages 174–191, 2003.

[35] Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 718–735. Springer, 2013.

[36] Josef Svenningsson and David Sands. Specification and verification of side channel declassification. In *FAST'09*, volume 5983 of *LNCS*, pages 111–125. Springer, 2009.

[37] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *SAS'2005*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.

[38] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 534–546. Springer, Heidelberg, April / May 2002.

[39] David J. Wheeler and Roger M. Needham. TEA, a tiny encryption algorithm. In Bart Preneel, editor, *FSE'94*, volume 1008 of *LNCS*, pages 363–366. Springer, Heidelberg, December 1995.

[40] Anna Zaks and Amir Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2008.

[41] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 503–516, 2015.

A Machine-Level Constant-Time Security

In this section we formalize constant-time security policies at the instruction set architecture (ISA) level. Modeling and verification at this low-level captures security of the actual executable code targeted by attacker scrutiny, which can differ significantly from the original source code prior to compilation and optimization. In order to describe our verification approach in a generic setting, independently of the computing platform and the nature of the exploits, we introduce abstract notions of *machines* and *observations*.

A.1 An Abstract Computing Platform

Formally, a *machine* $M = \langle A, B, V, I, O, F, T \rangle$ consists of

- a set A of address-space names,
- a set B of control-block names, determining the set $K = \{\texttt{fail}, \texttt{halt}, \texttt{spin}, \texttt{next}, \texttt{jump}(b) : b \in B\}$ of control codes,
- a set V of values — each address space $a \in A$ corresponds to a subset $V_a \subset V$ of values; together, the address spaces and values determine the set $S = A \to (V_a \to V)$ of states, each $s \in S$ mapping $a \in A$ to value store $s_a : V_a \to V$; we write $a{:}v$ to denote a reference to $s_a(v)$,
- a set I of instructions — operands are references $a{:}v$, block names $b \in B$, and literal values,
- a set O of observations, including the null observation ε,
- a leakage function $F : S \times I \to O$ determining the observation at each state-instruction pair, and
- a transition function $T : S \times I \to S \times K$ from states and instructions to states and control codes.

We assume that the value set V includes the integer value $0 \in \mathbb{N}$, that the control-block names include `entry`, and that the instruction set I includes the following:

$$T(s, \texttt{assume}\ a{:}v) = \begin{cases} \langle s, \texttt{spin} \rangle & \text{if } s_a(v) = 0 \\ \langle s, \texttt{next} \rangle & \text{otherwise} \end{cases}$$

$$T(s, \texttt{assert}\ a{:}v) = \begin{cases} \langle s, \texttt{fail} \rangle & \text{if } s_a(v) = 0 \\ \langle s, \texttt{next} \rangle & \text{otherwise} \end{cases}$$

$$T(s, \texttt{goto}\ b) = \langle s, \texttt{jump}(b) \rangle$$

$$T(s, \texttt{halt}) = \langle s, \texttt{halt} \rangle$$

We write $s[a{:}v_1 \mapsto v_2]$ to denote the state s' identical to s except that $s'_a(v_1) = v_2$.

Programs are essentially blocks containing instructions. Formally, a *location* $\ell = \langle b, n \rangle$ is a block name $b \in B$ and index $n \in \mathbb{N}$; the location $\langle \texttt{entry}, 0 \rangle$ is called the *entry location*, and L denotes the set of all locations. The location $\langle b, n \rangle$ is the *next successor* of $\langle b, n-1 \rangle$ when $n > 0$, and is the *start of block b* when $n = 0$. A *program for machine M* is a function $P : L \to I$ labeling locations with instructions.

A.2 Semantics of Abstract Machines

A *configuration* $c = \langle s, \ell \rangle$ *of machine M* consists of a state $s \in S$ along with a location $\ell \in L$, and is called

- *initial* when ℓ is the entry location,
- *failing* when $T(s, P(\ell)) = \langle \_, \texttt{fail} \rangle$,
- *halting* when $T(s, P(\ell)) = \langle \_, \texttt{halt} \rangle$, and
- *spinning* when $T(s, P(\ell)) = \langle \_, \texttt{spin} \rangle$.

The *observation* at c is $F(s, P(\ell))$, and configuration $\langle s_2, \ell_2 \rangle$ is the *successor* of $\langle s_1, \ell_1 \rangle$ when $T(s_1, P(\ell_1)) = \langle s_2, k \rangle$ and

- $k = \texttt{next}$ and ℓ_2 is the next successor of ℓ_1,
- $k = \texttt{jump}(b_2)$ and ℓ_2 is the start of block b_2, or
- $k = \texttt{spin}$ and $\ell_2 = \ell_1$.

We write $c_1 \to c_2$ when c_2 is the successor of c_1, and C denotes the set of configurations.

An *execution of program P for machine M* is a configuration sequence $e = c_0 c_1 \ldots \in (C^* \cup C^\omega)$ such that $c_{i-1} \to c_i$ for each $0 < i < |e|$, and $c_{|e|-1}$ is failing or halting if $|e|$ is finite, in which case we say that e is failing or halting, respectively. The *trace* of e is the sequence $o_0 o_1 \ldots$ of observations of at $c_0 c_1 \ldots$ concatenated, where $o \cdot \varepsilon = \varepsilon \cdot o = o$. Executions with the same trace are *indistinguishable*.

Definition 2 (Safety). *A program P on machine M is* safe *when no executions fail. Otherwise, P is* unsafe.

A.3 Constant-Time Security

To define our security property we must relate program traces to input and output values, since, generally speaking, the observations made along executions should very well depend on, e.g., publicly-known input values. Security thus relies on distinguishing program inputs and outputs as public or private. We make this distinction formally using address spaces, supposing that machines include

- a public input address space $\mathrm{i} \in A$, and
- a publicly observable output address space $\mathrm{o} \in A$,

in addition to, e.g., register and memory address spaces.

Intuitively, the observations made on a machine running a secure program should depend on the initial machine state only in the public input address space; when observations depend on non-public inputs, leakage occurs. More subtly, observations which are independent of public inputs can still be made, so long as each differing observation is eventually justified by differing publicly observable output values. Otherwise we consider that leakage occurs. Formally, we say that that two states $s_1, s_2 \in S$ are *a-equivalent* for $a \in A$ when $s_1(a)(v) = s_2(a)(v)$ for all $v \in V_a$. Executions e_1 from state s_1 and e_2 from state s_2 are *initially a-equivalent* when s_1 and s_2 are a-equivalent, and finite executions to s'_1 and s'_2 are *finally a-equivalent* when s'_1 and s'_2 are a-equivalent.

Definition 3 (Constant-Time Security). *A program is* secure *when:*

1. *Initially* i-*equivalent and finally* o-*equivalent executions are indistinguishable.*

2. *Initially* i-*equivalent infinite executions are indistinguishable.*

Otherwise, P is insecure.

The absence of publicly observable outputs simplifies this definition, since the executions of public-output-free programs are finally o-equivalent, trivially.

B Reducing Security to Safety

According to standard notions, security is a property over pairs of executions: a program is secure so long as executions with the same public inputs and public outputs are indistinguishable. In this section we demonstrate a reduction from security to safety, which is a property over single executions. The reduction works by instrumenting the original program with additional instructions which simulate two executions of the same program side-by-side, along the same control locations, over two separate address spaces: the original, along with a *shadow* of the machine state. In order for our reduction to be sound, i.e., to witness all security violations as safety violations, the control paths of the simulated executions must not diverge unless they yield distinct observations — in which case our reduction yields a safety violation. This soundness requirement can be stated via the following machine property, which amounts to saying that control paths can be leaked to an attacker.

Definition 4 (Control Leaking)**.** *A machine M is* control leaking *if for all states $s \in S$ and instructions $i_1, i_2 \in I$ the transitions $T(s,i_1) = \langle \_, k_1 \rangle$ and $T(s,i_2) = \langle \_, k_2 \rangle$ yield the same control codes $k_1 = k_2$ whenever the observations $F(s,i_1) = F(s,i_2)$ are identical.*

For the remainder of this presentation, we suppose that machines are control leaking. This assumption coincides with that of Section 4: all considered leakage models expose the valuations of branch conditions. Besides control leaking, our construction also makes the following modest assumptions:

- address spaces can be separated and renamed, and
- observations are accessible via instructions.

We capture the first requirement by assuming that programs use a limited set $A_1 \subset A$ of the possible address-space names, and fixing a function $\alpha : A_1 \to A_2$ whose range $A_2 \subset A$ is disjoint from A_1. We then lift this function from address-space names to instructions, i.e., $\alpha : I \to I$, by replacing each reference $a{:}v$ with $\alpha(a){:}v$. We capture the second requirement by assuming the existence of a function $\beta : I \times A \times V \to I$ such that

$$T(s, \beta(i,a,v)) = \langle s[a{:}v \mapsto F(s,i)], \texttt{next} \rangle.$$

For a given instruction $i \in I$, address space $a \in A$, and value $v \in V$, the instruction $\beta(i,a,v)$ stores the observation $F(s,i)$ in state $s \in S$ at $a{:}v$.

Following the development of Section 4, we develop an *output-insensitive* reduction which is always sound, but complete only for programs without publicly-annotated outputs. The extension to an *output-sensitive* reduction which is both sound and complete for all programs mirrors that developed in Section 4. This extension is a straightforward adaptation of Section 4's from high-level structured programs to low-level unstructured programs, thus we omit it here.

Assume machines include a vector-equality instruction

$$T(s, \texttt{eq}\ a_x{:}\vec{x}\ a_y{:}\vec{y}\ a{:}z) = \langle s[a{:}z \mapsto v], \texttt{next} \rangle$$

where $\vec{x}$ and $\vec{y}$ are equal-length vectors of values, and $v = 0$ iff $s(a_x)(x_n) \neq s(a_y)(y_n)$ for some $0 \leq n < |\vec{x}|$. This requirement is for convenience only; technically only a simple scalar-equality instruction is necessary.

To facilitate the checking of initial/final range equivalences for security annotations we assume that a given program P has only a single `halt` instruction, and

$$\begin{aligned} P(\texttt{entry}, 0) &= \texttt{goto}\ b_0 \\ P(\texttt{exit}, 0) &= \texttt{halt}. \end{aligned}$$

This is without loss of generality since any program can easily be rewritten in this form. Given the above functions α and β, and a fresh address space a, the *shadow product* of a program P is the program $P_\times$ defined by an entry block which spins unless the public input values in both i and α(i) address spaces are equal,

$$P_\times(\texttt{entry}, n) = \begin{cases} \texttt{eq i}{:}V_\text{i}\ \alpha(\text{i}){:}V_\text{i}\ a{:}x & n = 0 \\ \texttt{assume}\ a{:}x & n = 1 \\ \texttt{goto}\ b_0 & n > 1 \end{cases}$$

an exit block identical to the original program,

$$P_\times(\texttt{exit}, n) = P(\texttt{exit}, n)$$

and finally a rewriting of every other block $b \notin \{\texttt{entry}, \texttt{exit}\}$ of P to run each instruction on two separate address spaces,

$$P_\times(b, n) = \begin{cases} \beta(i,a,x) & n = 0 \pmod 6 \\ \beta(\alpha(i),a,y) & n = 1 \pmod 6 \\ \texttt{eq}\ a{:}x\ a{:}y\ a{:}z & n = 2 \pmod 6 \\ \texttt{assert}\ a{:}z & n = 3 \pmod 6 \\ i & n = 4 \pmod 6 \\ \alpha(i) & n = 5 \pmod 6 \end{cases}$$
$$\text{where } i = P(b, n/6)$$

while asserting that the observations of each instruction are the same along both simulations.

Theorem 3. *A safe program P with (respectively, without) public outputs is secure if (respectively, iff) $P_\times$ is safe.*