The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks

David Molnar[1], Matt Piotrowski[1], David Schultz[2], and David Wagner[1]

[1] UC-Berkeley {dmolnar, pio, daw}@eecs.berkeley.edu
[2] MIT das@csail.mit.edu

Abstract. We introduce new methods for detecting control-flow side channel attacks, transforming C source code to eliminate such attacks, and checking that the transformed code is free of control-flow side channels. We model control-flow side channels with a *program counter transcript*, in which the value of the program counter at each step is leaked to an adversary. The program counter transcript model captures a class of side channel attacks that includes timing attacks and error disclosure attacks.

We further show that the model formalizes previous *ad hoc* approaches to preventing side channel attacks. We then give a dynamic testing procedure for finding code fragments that may reveal sensitive information by key-dependent behavior, and we show our method finds side channel vulnerabilities in real implementations of IDEA and RC5, in binary modular exponentiation, and in the lsh implementation of the ssh protocol.

Further, we propose a generic source-to-source transformation that produces programs provably secure against control-flow side channel attacks. We implemented this transform for C together with a static checker that conservatively checks x86 assembly for violations of program counter security; our checker allows us to compile with optimizations while retaining assurance the resulting code is secure. We then measured our technique's effect on the performance of binary modular exponentiation and real-world implementations in C of RC5 and IDEA: we found it has a performance overhead of at most 5× and a stack space overhead of at most 2×. Our approach to side channel security is practical, generally applicable, and provably secure against an interesting class of side channel attacks.

1 Introduction

The last decade has seen a growing realization that side channel attacks pose a significant threat to the security of both embedded and networked cryptographic systems. The issue of information leakage via covert channels was first described by Lampson [29] in the context of timesharing systems, but the implications for cryptosystem implementations were not recognized at the time. In his seminal paper, Kocher showed that careful timing measurements of RSA operations could be used to discover the RSA private key through "timing analysis" [26]. Kocher, Jaffe, and Jun showed how careful power measurements could reveal private keys through "power analysis" [27]. Since then, side channel attacks have been used to break numerous smart card implementations of both symmetric and public-key cryptography [11, 32, 31, 33]. Later, Boneh and Brumley showed that a timing attack could be mounted against a remote web server [10]. Other recent attacks on SSL/TLS web servers make use of bad version oracles or padding check oracles; remote timing attacks can reveal error conditions enabling such attacks even if no explicit error messages are returned [30, 6, 43, 7, 24, 25, 35].

Defending against side channels requires a combination of software and hardware techniques. We believe a principled solution should extend the hardware/software interface by disclosing the side-channel properties of the hardware. Just as an instruction set architecture specifies the behavior of the hardware to sufficient fidelity that a compiler can rely on this specification, for side-channel security we advocate that this architecture should specify precisely what information the hardware might leak when executing by each instruction. This boundary forms a "contract" between hardware and software countermeaures as to who will protect what; the role of the hardware is to ensure that what is leaked is nothing more than what is permitted by the instruction set specification, and the role of the software countermeasures is to ensure that the software can tolerate leakage of this information without loss of security.

Our main technical contribution is an exploration of one simple but useful contract, the *program counter transcript model*, where the only information the hardware leaks is the value of the program counter at each step of the computation. The intuition behind our model is that it captures an adversary that can see the entire control flow behavior of the program, so it captures a non-trivial class of side-channel attacks. This paper develops methods for detecting such attacks on C programs and shows how to secure software against these attacks using a C-to-C code transformation.

First, we show the program counter model captures an interesting class of attacks, including timing analysis, a form of simple power analysis, and error disclosure attacks. For example, the model captures an adversary attempting network timing attacks of the type demonstrated by Boneh and Brumley [10]. We give a formal definition of *program counter security*, which expresses the requirement that the control flow behavior of the program contains no information of use to the attacker. We motivate our model by illustrating how previously proposed *ad hoc* countermeasures against error disclosure side channel attacks produce software that is program counter secure.

The program counter model enables testing for potential side channel vulnerabilities. The side channel behavior of a code fragment may leak information about the secret; therefore we would like to find and eliminate such fragments. We demonstrate a dynamic testing tool that searches for vulnerabilities in C code, and we show how the tool discovers real vulnerabilities in implementations of the RC5 and IDEA block ciphers, in binary modular exponentiation, and the lsh implementation of the ssh protocol. Once we have found a potential vulnerability, we want to eliminate it in a principled way.

We introduce a source-to-source program transformation that takes a program P and produces a transformed program P' with the same input-output behavior and with a guarantee that P' will be program counter secure. We built a prototype implementation of this transformation that works on C source code. We applied our implementation to implementations of RC5 and IDEA written in C, as well as an implementation of binary modular exponentiation. The resulting code is within a factor of at most 5 in performance and a factor of 2 in stack usage of untransformed code (§ 5.2).

Because our transform works at the C source level, we must be careful that the compiler does not break our security property. We build a static analysis tool that conservatively checks x86 assembly code for violations of program counter security. For example, we were able to show that our transformed code, when compiled with the Intel optimizing C compiler, retains the security properties.

The program counter model does not cover all side channel attacks. In particular, data dependent side channels (such as DPA or cache timing attacks [5]) are *not* eliminated by our transform. Nevertheless, we still believe the model is of value. We do not expect software countermeasures alone to solve the problem of side channels.

Instead, the program counter model states a contract: given that the hardware leaks only program counters, we can use our software transform to obtain security against side channel attacks. Designing hardware that meets this contract becomes itself an intriguing open problem, which can draw on the growing library of hardware side channel countermeasures. Furthermore, our framework is more general that the program counter transcript model. By varying the transcript model, we can shift more of the burden to hardware or software as needed. In Appendix A we sketch alternative transcript models.

Our transform for the program counter model provides a principled and pragmatic foundation for defending against an interesting class of side channel attacks. Our results show that, for all but the most performance-conscious applications, it may be possible to defend against control-flow side channels using a generic and automatic transformation instead of developing application-specific defenses by hand. Furthermore, we give static and dynamic methods for discovering side channel attacks in the program counter model. These methods generalize to alternative transcript models, as well.

In short, we show how to discover and defend against a class of attacks, while leaving defenses against some important attacks as an open question. Our work opens the way to exploring a wide range of options for the the interface between hardware and software side channel countermeasures, as

formalized by different transcript models. As such, our work is a first step towards efficient, principled countermeasures against side channel attacks.

2 A Transcript Model for Side Channel Attacks

We formalize the notion of side information by a *transcript*. We view program execution as a sequence of n steps. A transcript is then a sequence $T = (T_1, \ldots, T_n)$, where T_i represents the adversary's observation of the side channel at the i^{th} step of the computation. We will then write $P_k(x)$ to mean the program P running on secret input k and non-secret input x. Informally, a program is secure if the adversary "learns nothing" about k even given access to the side-channel transcript produced during execution of $P_k(x)$ for x values of its choice. Our model can be thought of as a "single-program" case of the Micali-Reyzin model, in which their "leakage function" corresponds to our notion of a transcript [34].

The transcript is the way we formalize the contract between hardware and software. It is the job of hardware designers to ensure that the hardware leaks nothing more than what is specified in the transcript, and the rest of this paper assumes that this has been done.

We write $D \sim D'$ if D and D' have the same distribution (perfect indistinguishability). Programs will take two inputs, a key k and an input x; we write $P_k(x)$ for the result of evaluating P on key k and input x. Define $\#P_k(x)\# \stackrel{\text{def}}{=} (y, T)$, where $y = P_k(x)$ is the result of executing P on (k, x) and T denotes the transcript produced during that execution. If P is randomized, $P_k(x)$ and $\#P_k(x)\#$ are random variables. We abuse notation and write $\#P_k\#$ for the map $\#P_k\#(x) = \#P_k(x)\#$. We can then define *transcript security* as follows:

Definition 1 *(transcript security). A program P is said to be transcript-secure (for a particular choice of transcript) if for all probabilistic polynomial time adversaries A, there exists a probabilistic polynomial time simulator S, which for all keys k satisfies $S^{P_k} \sim A^{\#P_k\#}$.*

Remark 1. In our model, the attacker is assumed to know the text of the program P. In fact, we allow the adversary A to depend on P in any way whatsoever. Thus, we do not consider any security that might be obtained through use of secret programs ("security through obscurity"); security must instead be obtained through the secrecy of the key k [23].

Remark 2. We could weaken the definition by letting $\sim$ denote statistical or computational indistinguishability, instead of perfect indistinguishability (as used in the formulation above). Because our schemes satisfy the stronger notion, our formulation above uses perfect indistinguishability, but the weaker notions are also meaningful. Further, the definition stated above requires that the simulation succeed for all keys k. Alternately, one could consider a weaker notion that requires only that simulation succeed for all but a negligible fraction of keys: e.g., for all A there exists S such that $(k, S^{P_k}) \sim (k, A^{\#P_k\#})$. This weaker notion would also be meaningful; however, since our schemes achieve the strong definition given above, we focus on the definition in its strongest form.

3 Program Counter Security: Security Against Certain Side-Channel Attacks

The notion of program counter security (*PC-security*) is motivated by the following side channel attacks:

Timing attacks. In a timing attack, the attacker learns the total time taken by the computation. Thus, the side-channel transcript T seen by the adversary might contain a single value representing the time taken during execution of the program. Any program that is transcript-secure with this definition of transcript will then be secure against timing attacks.

Simple power analysis. We could also consider leakage due to the difference in power consumption between different types of machine instructions. For instance, a MUL (multiply) instruction might

consume more power than an XOR instruction; a LD (load) instruction might have a different pattern of power consumption than a CMP (comparison). This suggests a model of security where the adversary is permitted to see the opcode (type of instruction) executed at each step, but not the value of its operands.

We observe that these models are concerned mainly with the control-flow behavior of the program. In each case, the side channel leaks partial information about the path the processor takes through the program, but nothing about the data values. Consequently, these models can be subsumed by a single, more general model of security, which we call the *program counter model* or *PC model*.

In the PC model, the transcript T conveys the sequence of values taken on by the processor's program counter during the computation. To be specific, our concrete notion of security is as follows:

Definition 2 *(PC-security). A program P is* PC-secure *if P is transcript-secure when the transcript $T = (T_1, \ldots, T_n)$ is defined so that T_i represents the value of the program counter at the i^{th} step of the execution.*

Consequently, in the PC model, the attacker learns everything about the program's control-flow behavior but nothing about other intermediate values computed during execution of the program. In the remainder of this work, we make two assumptions about the hardware used to execute the program: first, that the the program text is known to the attacker. This implies that the program counter at time i reveals the opcode that was executed at time i. Second, the side-channel signal observed by the attacker depends only on the sequence of program counter values, e.g., on the opcode executed. For example, we assume that the execution time of each machine instruction can be predicted without knowledge of the values of its operands, so that its timing behavior is not data-dependent in any way. Warning: This is not true on some architectures, due to, among other things, cache effects [5], data-dependent instruction timing, and speculation.

We stress that our transcript model is intended as an idealization of the information leaked by the hardware; it may be that no existing system meets the transcript precisely. Nonetheless, we believe these assumptions are reasonable for some embedded devices, namely those which do not have caches or sophisticated arithmetic units. With these assumptions, PC-security subsumes the attacks mentioned above. Given a transcript of PC values, the attacker can infer the total number of machine cycles used during the execution of the program, and thus the total time taken to run this program; consequently, any program that is PC-secure will also be secure against timing attacks.

3.1 Justifying Previous *Ad Hoc* Countermeasures

It is interesting to note that many heuristic defenses that have been previously proposed in the literature can be formally justified using the notion of PC-security. We give some examples.

Coron's table-lookup method. The simple square-and-multiply exponentiation algorithm shown in Fig. 1(a) is insecure against power analysis, because squaring leaves a different pattern of power consumption than multiplication. Coron proposed a method using table lookups to defend against this attack [15]; see Fig. 1(b). The security of Coron's method rests on similar assumptions to those formalized in PC-security: the power consumption of all operations must be independent of their operands.

We note that Coron's method can be readily justified by our notion of PC-security. Indeed, Coron's modified exponentiation algorithm is PC-secure (given an appropriate implementation of the multiplication and squaring subroutines), since the sequence of program counter values is independent of the key k.

Universal exponentiation. Clavier and Joye proposed a new exponential algorithm. Their algorithm operates as an interpreter for a very simple register-based language, executing a sequence of instructions $\Gamma = (\Gamma_1, \ldots, \Gamma_\ell)$ written in this language in a manner secure against side channels [14]. In their language,

```
Exponentiate(k, x):
1. y ← 1; ℓ ← len(k)
2. for i ← ℓ − 1, ..., 0:
3.    if k_i = 1
4.       then y ← y^2 × x
5.       else  y ← y^2
6. return y
```

(a) The insecure version.

```
Coron-Exp(k, x):
1. y ← 1; ℓ ← len(k)
2. for i ← ℓ − 1, ..., 0:
3.    A[1] ← y^2 × x
4.    A[0] ← y^2
5.    y ← A[k_i]
6. return y
```

(b) Coron's method.

```
Universal-Exp(Γ, x):
1. Parse Γ as (Γ_1, ..., Γ_ℓ)
2. Parse each Γ_i as (γ_i : α_i, β_i)
3. R[α_1] ← x; R[β_1] ← x
4. for i ← 1, ..., ℓ:
5.    R[γ_i] ← R[α_i] × R[β_i]
6. return R[γ_ℓ]
```

(c) Clavier-Joye's method.

Fig. 1. Three methods for computing x^k in some (unspecified) group. Method (a) is insecure, because of the key-dependent if statement. Method (b) uses a table lookup to avoid a key-dependent branch, while method (c) interprets a small program γ that takes time independent of the key k.

```
a = a << 1;
if (carry)
   a = a ^ 0x1B;
```

(a) Naïve.

```
a = a << 1;
b = a;
a = a - (a + carry);
a = a & 0x1B;
a = a ^ b;
```

(b) SKKS.

```
b = a;
b = b << 1;
a = a / 128;
a = a * 0x1B;
a = a ^ b;
```

(c) BS #1.

```
f = a >> 7;
a = a << 1;
x[0] = a ^ 0x00;
x[1] = a ^ 0x1B;
a = x[f];
```

(d) BS #2.

Fig. 2. Four implementations of the AES `xtime` operation, intended for 8-bit microprocessors. The first is insecure against timing and DPA attacks. The next three have been used by implementors to defend against such attacks. Note: the instruction '`a << 1`' sets the `carry` bit if the most significant bit of `a` is set.

there are m registers $R[1], \ldots, R[m]$, and every instruction has the form $\Gamma_i = (\gamma_i : \alpha_i, \beta_i)$, which is interpreted as $R[\gamma_i] \leftarrow R[\alpha_i] \times R[\beta_i]$. See Fig. 1(c). They, too, assume that power consumption is not data-dependent in any way. Because the sequence of program counter values depends only on ℓ and not on Γ, the Clavier-Joye algorithm is PC-secure assuming that ℓ is public and that the multiplication and squaring subroutines are implemented appropriately.

In other words, their algorithm takes as input an addition chain $\Gamma(k)$ (for exponentiation to the k^{th} power) and a base x and applies the exponentiation chain securely to compute x^k in a way defined so that side channels will not reveal k. See Fig. 1(c). Assuming that ℓ, the length of the addition chain, is public, their algorithm is PC-secure (given an appropriate implementation of the multiplication and squaring subroutines), since the sequence of program counter values depends only on ℓ and not on Γ.

The AES `xtime` *primitive.* One of the main operations used in AES is the `xtime` function, which corresponds to shifting a 8-bit LFSR left one bit (or, equivalently, multiplying by '`02`' in $GF(2^8)$). It is well-known that the straightforward implementation of `xtime` (see Fig. 2(a)) is susceptible to timing and differential power analysis (DPA) attacks, because it uses a conditional jump instruction [28]. Sano, et al., proposed an alternate implementation that runs in constant time and hence is secure against timing attacks [40] (See Figure 2(b)), and Blömer and Seifert described two more [9] (See Figure 2(c) and (d)). We note that all three alternate implementations can be readily shown to be PC-secure, providing further justification for these countermeasures.

Error disclosure attacks. Some implementation attacks exploit information leaks from the disclosure of decryption failures. Consider a decryption routine that can return several different types of error messages, depending upon which stage of decryption failed (e.g., improper PKCS formatting, invalid padding, MAC verification failure). It turns out that, in many cases, revealing which kind of failure occurred leaks information about the key [6, 43, 7, 24, 25, 35].

OAEP-INSECURE(d, x):
1. $(e, y) \leftarrow$ INTTOOCTET$(x^d \bmod n)$
2. **if** e **then return** Error
3. $(e', z) \leftarrow$ OAEPDECODE(y)
4. **if** e' **then return** Error
5. **return** z

(a) Naïve code (insecure).

OAEP-SECURE(d, x):
1. $(e, y) \leftarrow$ INTTOOCTET$(x^d \bmod n)$
2. $y \leftarrow$ COND$(e, \text{dummy value}, y)$
3. $(e', z) \leftarrow$ OAEPDECODE(y)
4. **return** COND$(e \vee e', \mathsf{Error}, z)$

(b) A transformed version (PC-secure).

Fig. 3. Two implementations of OAEP decryption. We assume that each subroutine returns a pair (e, y), where e is a boolean flag indicating whether any error occurred, and y is the value returned by the subroutine if no error occurred. The code on the left is insecure against Manger's attack, because timing analysis allows to distinguish an error on Line 2 from an error on Line 3. The code in the right is PC-secure and hence not vulnerable to timing attacks, assuming that the subroutines are themselves implemented in a PC-secure form.

Naïvely, one might expect that attacks can be avoided if the implementation always returns the same error message, no matter what kind of decryption failure occurred. Unfortunately, this simple countermeasure does not go far enough. Surprisingly, in many cases timing attacks can be used to learn which kind of failure occurred, even if the implementation does not disclose this information explicitly [6, 43, 7, 24, 25, 35, 30]. See, for instance, Fig. 3(a). The existence of such attacks can be viewed as a consequence of the lack of PC-security. Thus, a better way to defend against error disclosure attacks is to ensure that all failures result in the same error message and that the implementation is PC-secure.

Suppose we had a subroutine COND(e, t, f) that returns t or f according to whether e is true or false. Using this subroutine, we propose in Fig. 3(b) one possible implementation strategy for securing the code in Fig. 3(a) against error disclosure attacks. If there is an error in Line 1, we generate a dummy value (which can be selected arbitrarily from the domain of OAEPDECODE) to replace the output of Line 1. If all subroutines are implemented in a PC-secure way and we can generate a dummy value in a PC-secure way, then our transformed code will be PC-secure and thus secure against error disclosure attacks. There is one challenge: we need a PC-secure implementation of COND. We propose one way to meet this requirement through logical masking:

COND(e, t, f):
1. $m \leftarrow$ MASK(e)
2. **return** $(m \wedge t) \vee (\neg m \wedge f)$

Here $\neg, \vee, \wedge$ represent the bitwise logical negation, OR, AND (respectively). This approach requires a PC-secure subroutine MASK satisfying MASK$(\mathsf{false}) = 0$ and MASK$(\mathsf{true}) = 2^\ell - 1 = 11 \cdots 1_2$, assuming t and f are ℓ-bit values. The MASK subroutine can be implemented in many ways. For instance, assuming true and false are encoded as values 1 and 0, we could use MASK$(e) = (2^\ell - 1) \times e$; MASK$(e) = -e$ (on two's-complement machines); MASK$(e) = (e \ll (\ell - 1)) \ggg (\ell - 1)$ (using a sign-extending arithmetic right shift); or several other methods. With the natural translation to machine code, these instantiations of MASK and COND will be PC-secure.

3.2 Straight-Line Code is PC-Secure

The key property we have used to show PC-security of code in the previous section is that the code is *straight-line*, by which we mean that the flow of control through the code does not depend on the data in any way. We now encapsulate this in a theorem.

Theorem 1. *(PC-security of straight-line code). Suppose the program P has no branches, i.e., it has no instructions that modify the PC. Then P is PC-secure.*

```
CONST static uint16 mul(register uint16 a, register uint16 b) {
    register word32 p;

    p = (word32) a * b;
    if (p) {
        b = low16(p);
        a = p >> 16;                              mean   stddev    min     max
        return (b - a) + (b < a);                 27.00   0.03      26      27
    } else if (a) {
        return 1 - a;                             7.00    0.02       7       8
    } else {
        return 1 - b;                             0.00    0.02       0       1
    }
}
```

Fig. 4. PGP's implementation of multiplication mod $2^{16}+1$. This routine is called a total of 34 times per IDEA operation. The numbers on the right show the mean, standard deviation, minimum, and maximum number of times each of the corresponding source code lines is visited per IDEA call over $10,000$ random keys and a fixed plaintext. The non-zero standard deviation indicates these lines as potential trouble spots, since it means the number of times these lines are executed varies as the secret input varies. Therefore, measuring a program's behavior on these lines may reveal information about the secret input.

Proof. Since P is branch-free, for all inputs x and all keys k the program counter transcript T of $P_k(x)$ will be the same. For any adversary A, consider the simulator S that runs A, outputting whatever A does and answering each query x that A makes to its oracle with the value $(P_k(x), T)$. Then $S^{P_k} \sim A^{\#P_k\#}$ for all k.

In fact, it suffices for P to be free of key-dependent branches. We can use this to show that some looping programs are PC-secure.

Theorem 2. *(PC-security of some looping programs). Suppose the program P consists of straight-line code and loops that always run the same code body for a fixed constant number of iterations in the same order (i.e., the loop body contains no break statements and no assignments to the loop counter; there are no* `if` *statements). Then P is PC-secure.*

Proof. As before, for all inputs x and all keys k, the program counter transcript T of $P_k(x)$ will be the same, so we can use the same simulation strategy.

4 Finding Side Channel Attacks With Run-time Profiling

Not only does PC-security give us a way to prove the security of code, but it can also guide us to potential attacks. By looking at the transcripts of programs on many different secret inputs, we can search for key-dependencies in the transcripts. In particular, if we find any input x and any pair of keys k, k' so that $P_k(x)$ yields a different program counter transcript than $P_{k'}(x)$, we can conclude that P is not PC-secure. Our algorithm, then is simple: pick an input x; pick many random keys $k_1, \ldots, k_n$; and look for any variation whatsoever in the resulting program counter transcripts. Variations indicate potential side channel vulnerabilities and thus demand further investigation.

We built a tool that automates this process. We use the the GNU gcov tool to measure how many times each line of code is executed during each execution. We execute the program many times with many different secret keys, and we compute statistics for each line. This allows us to detect whether there is any line of code that is executed more times with one key then with another, which indicates a violation of PC-security.

This tool is fairly crude, in several respects. First, it does not gather entire program counter transcripts, so it could in principle miss some violations of PC-security. Second, gcov does not handle optimized C code very well, so we focus mainly on unoptimized code. We could work around these limitations by using an emulator, such as Bochs or Valgrind, but we leave this for future work [13]. Still, as we will see, our tool gives us a simple yet powerful test for potential side channel vulnerabilities.

Case Study: IDEA. To gain experience with this approach, we applied our tool to the implementation of the International Data Encryption Algorithm (IDEA) block cipher in PGP, Inc.'s version of PGP 2.6.2. IDEA relies heavily on multiplication modulo $2^{16}+1$. However, this operation is tricky to implement in a constant-time way, because multiplying by zero must be treated as a special case. Others have shown the existence of timing attacks against some IDEA implementations based on this observation [22].

We decided to apply our tool to PGP's IDEA implementation. It quickly found several lines of code that were executed more times for some keys than others, indicating the possibility of a side-channel attack. See Fig. 4 for the tool's output. Upon investigating these key-dependencies, we noticed that these indicate a real vulnerability. In the case where either a or b is zero, the code takes a shortcut and returns in three instructions. If neither a nor b is zero, the code takes six instructions, leading to a timing difference. Thus, our tool sufficed to discover an instance of the aforementioned timing attack, even though our tool contains no special knowledge of that attack.

Upon further investigation, we discovered that these kind of timing dependencies are common in IDEA implementations. PGP 6.58 contains similar code. PGP 2.3a has an even worse timing difference; the multiplication is only carried out if both operands are non-zero.

Other case studies. Through similar methods, we discovered that the lsh implementation of the ssh protocol has different behavior when a username corresponds to a valid user and when it does not. This allows a remote attacker to guess usernames and confirm his guess by measuring the timing behavior of the server's response. We also found that the implementation of RC5 shipped with TinyOS is not PC-secure. TinyOS is an operating system designed for wireless sensor networks; the RC5 implementation we considered was part of the link-layer encryption scheme TinySEC, though it has since been deprecated and replaced by Skipjack due to patent concerns. These examples illustrate that violations of PC-security are easy to find with runtime profiling, and that they often correspond to side-channel vulnerabilities.

5 Code Transformation for PC-Security

The examples we have seen so far illustrate the relevance of PC-security, but enforcing PC-security by hand is highly application-dependent and labor-intensive. We describe a generic method for transforming code to be PC-secure. Given a program P, the transformed program P' is PC-secure and has the same input-output behavior as P on all inputs (i.e., $P_k(x) \sim P'_k(x)$ for all k, x). It may be surprising that almost all code can be transformed in this way, but we show in the Appendix how this can be done for loops where a fixed upper bound is known at compile time (i.e., primitive recursive function). We first explain informally how our transform works, and then we carefully prove its security.

Transforming conditional statements. An `if` statement containing only assignment expressions can be handled in a fairly simple way. To provide PC-security, we execute both branches of the `if`, but only retain the results from the branch that would have been executed in the original program. The mechanism we use to nullify the side-effects of either the consequent or the alternative is *conditional assignment*. We have already seen one way to implement PC-secure conditional assignment using logical masking and the COND subroutine. For example, the C statement `if (p) { a = b; }` can be transformed

```
if (n % 2) {
    r = r * b;
    n = n - 1;
} else {
    b = b * b;
    n = n / 2;
}
```

```
m = -(n % 2);
 r = (m & (r * b)) | (~m & r);
 n = (m & (n - 1)) | (~m & n);
m = ~m;
 b = (m & (b * b)) | (~m & b);
 n = (m & (n / 2)) | (~m & n);
```

Fig. 5. Transforming a simple `if` statement to ensure PC-security. On the left, the original, insecure version. On the right, the result of our automatic transformation.

```
if (n != 0) {
    if (n % 2) {
        r = r * b;
        n = n - 1;
    } else {
        b = b * b;
        n = n / 2;
    }
}
```

```
m1 = -(n != 0);
 m2 = m1 & (-(n % 2));
  r = (m2 & (r * b)) | (~m2 & r);
  n = (m2 & (n - 1)) | (~m2 & n);
 m2 = m1 & ~m2;
  b = (m2 & (b * b)) | (~m2 & b);
  n = (m2 & (n / 2)) | (~m2 & n);
```

Fig. 6. An example of our transformation applied to a nested `if` statement.

to `a = COND(m, b, a)`, where `m = MASK(p)`. If `p` represents a 0-or-1-valued boolean variable[3], this might expand to the C code `m = -p; a = (m & b) | (~m & a)`.

Fig. 5 shows a complete transformation for part of an iterative exponentiation routine.

Many architectures have conditional instructions that support conditional assignment more naturally. For instance, virtually any ARM instruction can be made conditional on a processor flag, and the Pentium Pro and later IA-32 processors have a conditional `mov` instruction. On these architectures, our transform can be implemented more efficiently, but our prototype currently does not attempt to take advantage of native support for conditional assignment.

Nested `if` statements are only slightly more involved. Instead of assigning the mask based solely on the value of the predicate, we use a conditional assignment that depends on the value of the mask for the surrounding `if`. We conditionally invert the mask before the code for the `else` clause based on the enclosing `if`'s mask. This means that any given statement in the program only needs to care about a single mask for `if`s, regardless of the level of nesting.

Loops. Loops present difficulties because the number of iterations they will perform may not be known statically, and in particular, the number of iterations may depend on sensitive data. Fortunately, in many cases a constant upper bound can be established on the number of iterations. We transform the loop so that it always executes for the full number of iterations, but the results of any iterations that would not have occurred in the original program are discarded. A specification of our entire transform may be found in Section 5.1

More generally, it would suffice to know that the number of iterations can be upper-bounded by a value that depends only on public inputs (even if this bound is not constant). A future version of our transform might work around this by performing static analysis to determine which loops actually depend on secret data and which do not. We found, however, that for the examples we looked at such an analysis was not necessary.

Fortunately, it is frequently the case, particularly in cryptographic algorithms, that the number of iterations a loop performs is known statically or depends only on non-sensitive data. In these situations, we can leave the loop unchanged. A more interesting scenario is when a loop may execute for a variable number of iterations, but we can establish a reasonably tight upper bound on the number of iterations.

[3] If `p` is not guaranteed to be 0-or-1-valued, this definition of `m` does not work. We use `m = !p - 1` instead.

$C \in \text{Com} = \text{Program}$
$E \in \text{Exp}$
$B \in \text{BoolExp} \subset \text{Exp}$
$I \in \text{Identifier}$
$\text{arithop} \in \text{AOp} = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{\&}, \texttt{|}\}$
$\text{relop} \in \text{RelOp} = \{\texttt{>}, \texttt{<}, \texttt{=}\}$
$\text{boolop} \in \text{BoolOp} = \{\texttt{and}, \texttt{or}\}$
$n \in \text{Num}$

(a) Syntactic domains.

$$C ::= I \texttt{ := } E \mid C'\texttt{; } C'' \mid \texttt{if } B \texttt{ then } C' \texttt{ else } C'' \mid \texttt{for } I \texttt{ := } n \texttt{ to } n' \texttt{ do } C' \mid \texttt{break}$$

$$E ::= I \mid n \mid B \mid E' \text{ arithop } E'' \mid \texttt{\~{}}E'$$

$$B ::= 0 \mid 1 \mid B' \text{ boolop } B'' \mid E' \text{ relop } E'' \mid \texttt{!}B'$$

(b) Grammar.

Fig. 7. The abstract syntax of IncredibL.

For instance, the loop may contain a `break` statement intended to cause it to exit early in some cases, making the program insecure in the PC model. We transform the loop so that it always executes for the full number of iterations, but the results of any iterations that would not have occurred in the original program are discarded. In essence, this can be thought of as prepending `if (`*loop_done*`)` to every statement in the loop, and transforming `break` statements into conditional assignments to *loop_done*.

5.1 Specifying The Transform

With these examples, we are now ready to specify the transform more precisely. As we discuss below, our implementation handles all of the C language. However, the lack of a formal semantics for C makes it difficult to prove anything about C, so we focus on a subset of C that contains most of the language features that are relevant to our analysis. For this subset, we can prove that the transform is semantically preserving and that it produces PC-secure code.

To precisely capture this subset of C, we introduce IncredibL, a simple imperative language with restricted control flow. IncredibL is our own invention, but it is derived from Hennessy's WhileL [19]. The grammar for IncredibL can be found in Fig. 7.

Roughly, IncredibL captures a memory-safe subset of C with only bounded loops, `if` statements, and straight-line assignments. Note that we do not allow any forms of recursion or unstructured control flow, as these may introduce unbounded iteration. We also disallow calls to untransformed subroutines, including I/O primitives. Note that because loop bounds are known statically in IncredibL, we can in principle unroll all loops in any IncredibL program to obtain code with no branches.

Our transformation $\mathcal{T}_{\text{Program}}$ is specified in Fig. 8. We state the main theorems here. Proofs may be found in the Appendix.

Theorem 3. *$\mathcal{T}_{Program}$ is semantics-preserving: for every valid IncredibL program P and every pair of inputs k, x, $P'_k(x) \sim P_k(x)$, where $P' \stackrel{def}{=} \mathcal{T}_{Program}[\![P]\!]$.*

Theorem 4. *$\mathcal{T}_{Program}$ outputs safe code: for every IncredibL program P, $\mathcal{T}_{Program}[\![P]\!]$ consists only of straight-line code and loops with straight-line code bodies that run for a fixed constant number of iterations with no assignments to induction variables.*

Corollary 1. *$\mathcal{T}_{Program}$ enforces PC-security: for every IncredibL program P, $\mathcal{T}_{Program}[\![P]\!]$ is PC-secure.*

$$
\begin{aligned}
\mathcal{T}_{\text{Program}}[\![C]\!] &= I_0 \texttt{ := -1; } \mathcal{T}_{\text{Com}}[\![C]\!](I_0, I_0) \quad \text{where } I_0 \text{ is a fresh identifier} \\
\mathcal{T}_{\text{Com}}[\![I \texttt{ := } E]\!](I_{\text{if}}, I_{\text{brk}}) &= \textit{conditional-assign}(I, \texttt{(}I_{\text{if}} \texttt{ \& } I_{\text{brk}}\texttt{)}, E, I) \\
\mathcal{T}_{\text{Com}}[\![C\texttt{; } C']\!](I_{\text{if}}, I_{\text{brk}}) &= \mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}})\texttt{; } \mathcal{T}_{\text{Com}}[\![C']\!](I_{\text{if}}, I_{\text{brk}}) \\
\mathcal{T}_{\text{Com}}[\![\texttt{if } B \texttt{ then } C \texttt{ else } C']\!](I_{\text{if}}, I_{\text{brk}}) &= \textit{conditional-assign}(I_0, \texttt{(}I_{\text{if}} \texttt{ \& } I_{\text{brk}}\texttt{)}, \texttt{(0-}B\texttt{)}, \texttt{0})\texttt{; } \mathcal{T}_{\text{Com}}[\![C]\!](I_0, I_{\text{brk}})\texttt{;} \\
&\quad \textit{conditional-assign}(I_0, \texttt{(}I_{\text{if}} \texttt{ \& } I_{\text{brk}}\texttt{)}, \texttt{\~{}}I_0, \texttt{0})\texttt{; } \mathcal{T}_{\text{Com}}[\![C']\!](I_0, I_{\text{brk}}) \\
&\quad \text{where } I_0 \text{ is a fresh identifier} \\
\mathcal{T}_{\text{Com}}[\![\texttt{for } I \texttt{ := } n \texttt{ to } n' \texttt{ do } C]\!](I_{\text{if}}, I_{\text{brk}}) &= \textit{conditional-assign}(I_0, \texttt{(}I_{\text{if}} \texttt{ \& } I_{\text{brk}}\texttt{)}, \texttt{-1}, \texttt{0})\texttt{;} \\
&\quad \texttt{for } I \texttt{ := } n \texttt{ to } n' \texttt{ do } \mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_0) \\
&\quad \text{where } I_0 \text{ is a fresh identifier} \\
\mathcal{T}_{\text{Com}}[\![\texttt{break}]\!](I_{\text{if}}, I_{\text{brk}}) &= \textit{conditional-assign}(I_{\text{brk}}, \texttt{(}I_{\text{if}} \texttt{ \& } I_{\text{brk}}\texttt{)}, \texttt{0}, I_{\text{brk}}) \\
\textit{conditional-assign}(I, E_m, E_t, E_f) &= I \texttt{ := (}E_t \texttt{ \& } E_m\texttt{) | (}E_f \texttt{ \& \~{}}E_m\texttt{)}
\end{aligned}
$$

Fig. 8. A formal specification of our transform.

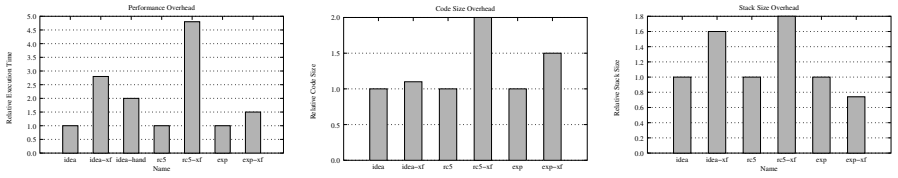

Fig. 9. The speed, code size, and stack size overhead of our transform, as applied to modular exponentiation, IDEA, and RC5. The -xf suffix indicates the automatic application of our transform, while the -hand suffix indicates a hand-optimized application of our transform. Values are normalized: the untransformed version of a program takes unit time by definition, while the transformed version of the same program is shown with the relative multiplicative overhead compared to the untransformed version.

5.2 Transform Implementation

We applied our transform to implementations of the IDEA and RC5 block ciphers, which are known to be susceptible to timing attacks unless implemented carefully [22, 18]. We also applied our transform to a simple binary modular exponentiation implementation built on top of the GNU Multiprecision Library. We chose x86 as our reference platform due to its widespread popularity, and we used the Intel C Compiler, Version 8.1 for our performance results. Our tests were run on a 2.8 GHz Pentium 4 running FreeBSD 6-CURRENT (April 17, 2005).

We first optimized our transform by hand on IDEA's multiplication routine to determine how fast our transform can be in principle. Our hand-optimized transformation achieves a factor of 2× slowdown compared to untransformed code, when both are compiled using the Intel C compiler with `-O3`.

We then implemented an automatic C source-to-source transformation using the C Intermediate Language package [36]. Our implementation was intended as an early prototype to demonstrate the feasibility of applying our transformation automatically. With more careful attention, better performance from an automatic transform may be possible.

Performance results. Our performance results for modular exponentiation, IDEA, and RC5 are presented in Fig. 9. For IDEA, we transformed only the `mul` routine, which we identified as the main candidate for timing and power attacks. For RC5, we performed the transformation on the rotate routine, for similar reasons. For modexp, we transformed only the main loop, but did not transform GnuMP library functions.

The performance of untransformed, optimized code is set to 1, and the performance of transformed code is reported relative to this value; for example, the bar with height "2" for `idea-hand` indicates

```
not_pc_secure:
 1         movl      $1, %eax
 2         movl      $42, %edx
 3         cmpl      $100, %eax
 4         jge       out
 5 loop:   imull     %eax, %edx
 6         addl      4(%esp), %eax
 7         cmpl      $100, %eax
 8         jl        loop
 9 out:    movl      %edx, %eax
10         ret
```

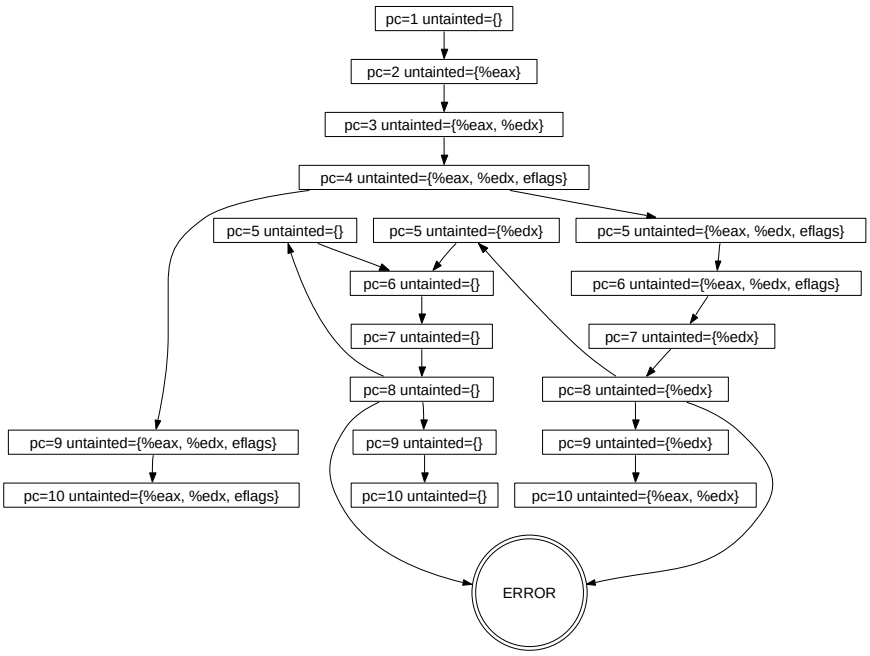

Fig. 10. Example: State space explored by the static verifier.

that our hand-transformed IDEA code took 2 times as long as untransformed code. We also found that code size increased by at most a factor of 2. Finally, we considered the stack usage of transformed code; this is the most relevant metric of memory usage for systems without dynamically allocated memory.

We can see that both our automatic transform is within a factor of 5 in performance of the optimized untransformed code in all cases. Again, our implementation is a prototype intended to test feasibility; with more work, more efficient code may be possible. Further, our stack size and code size increase by at most a factor of 2, showing that the transformed code may still be reasonable even in constrained environments. Our results suggest that a fully automatic transformation with slowdown acceptable for many applications is possible with care.

A static analysis for PC-security. We cannot guarantee that the compiler will preserve our transform's straight-line + restricted-loop guarantee when it generates assembly language. We addressed this problem by building a simple static checker for x86 assembly code that detects violations of PC-security. If the compiler does introduce assembly constructs that violate PC-security, these constructs will be flagged by the checker. We can then revise the code or improve our transform. Our checker is sound, but not complete: it will catch all violations of PC-security, but may also report false positives.

In fact, our checker caught unsafe constructs in the gcc 3.3.2 compilation of our transformed C code to x86 assembly. In certain expression contexts, gcc compiles the logical negation (`!`) operator into an assembly sequence involving a conditional branch. Further experimentation reveals that this idiom is not limited to the x86; the Sun C compiler on an UltraSPARC-60 machine exhibits similar behavior. We discovered, however, that the Intel C compiler does not compile `!` using conditional jumps, so we used the Intel compiler for our performance experiments. One alternative would be to change the transform

to avoid the `!` operator, but we did not find a portable and efficient replacement. Another alternative would be to modify gcc to add an extra mode that respects the PC-security of compiled code; we found it easier, however, to simply use the Intel compiler for our tests. Our experience shows the merely turning off optimizations does not guarantee that transformed C code will be PC-secure after compilation.

The checker works by performing an information flow analysis on the control flow graph for the program. Register and memory locations are classified as either *sensitive* or *public*. A public value is one that is derived entirely from constants, whereas a sensitive value is not, and therefore may depend on secret data. The program is simulated on a nondeterministic abstract machine in which each state consists of a program counter and a set of public locations. State transitions are defined solely based on the input/output behavior of the instructions in the program, except for conditional branches, which are handled specially. For these branches, both branch directions are explored nondeterministically. Additionally, if branch predicate is sensitive, the machine transitions to an error state and flags a violation of PC-security.

For these branches, both branch directions are explored nondeterministically. Additionally, if branch predicate is sensitive, the machine transitions to an error state and flags a violation of PC-security.

This approach is sound but not complete; it identifies all actual violations of PC-security, but may also report false positives. False positives arise because some aspects of the execution environment (pointer aliasing in particular) are difficult or impossible to model accurately using our analysis, and so we adopt conservative approximations. Furthermore, the model does not capture the fact that some computations such as $x \oplus x$ produce a public output on sensitive inputs. Nevertheless, these false positives do not appear to be a problem in practice; we encountered none in the programs we examined. Since the verifier only needs to understand data and control flow and not actual instruction semantics, our implementation is relatively simple, consisting of about 1800 lines of Java code. Fig. 10 shows the state space explored on one possible translation of the following C program to x86 assembly.

```
int not_pc_secure(int x) {
    int res = 42;
    for (int i = 1; i < 100; i += x)
        res *= i;
    return res;
}
```

The program is not PC-secure because the execution of all loop iterations after the first are dependent upon the unknown input `x`. The second branch is flagged as a violation of PC-security because it depends on the processor's *less than* flag (part of the `eflags` register), which is sensitive at the branch point.

The state space diagram shows that the program is not PC-secure. We find the first branch does not violate PC-security because in the unique control path that reaches it, the flags that determine the branch direction are public. The tool considers both branch directions: the left path (which is infeasible in practice) skips the loop entirely and returns from the function, and the right path executes the loop body. In the rightmost path, `4(%esp)`, the potentially sensitive parameter to the function, is added to `%eax`, so `%eax` is sensitive. Hence, `eflags` is made sensitive by the comparison of `%eax` to 100 on line 7, and the branch on line 8 is flagged as a violation of PC-security.

For the sake of comprehensiveness, the tool also explores the center path in the figure. This path corresponds to the second and subsequent loop iterations, which have an even smaller set of public values than the first iteration. Note that even though there is no practical bound on the number of iterations of the loop in this example, the sensitive set stabilizes quickly, so the state space that must be explored is small.

6 Related Work

Many previous side channel defenses are application-specific. For example, blinding can be used to prevent timing attacks against RSA [26, 10]. The major advantage of an application-specific defense is that it can be efficient. Experimental measurements show that blinding only adds a 2–10% overhead; contrast this with the overhead we measured in § 5.2.

Unfortunately, no proof of security for blinding against side channel attacks is known. In the absence of proof, it is difficult to assess whether the defense works. For example, defenses were designed for the five AES finalists [31]. These defenses had no formal model of information leaked to the adversary and hence no way to verify security. In fact, Coron and Goubin later pointed out a subtle flaw in one of the techniques [16]. Blömer, et al., give several more examples of techniques that were thought to be secure but failed, and of cases where innocent-looking "simplifications" failed. These examples motivate their study of provably secure defenses against side channels [8]. We note that Hevia and Kiwi showed that conditional branches in some implementations of DES leak information about the secret key; this is another motivation for PC-security.

Chevallier-Mames, Ciet, and Joye show a clever construction for performing modular exponentiation without incurring undue overhead. They show that, under an appropriate physical assumption, only the Hamming weight of the exponent is leaked by their algorithm. Blömer, et al., also define a model for provable security against side channel attacks and show how to implement AES securely in this model [8]. While these methods are a step forward, they still require a great deal of new effort for each new application.

The programming languages community has studied the problem of secure information flow extensively, but most work regarding C code has focused on detecting covert channels and side channels [39], not on eliminating them via code transformation. One exception is Agat's work, which transforms out timing leaks by inserting dummy instructions to balance out the branches in a program's control-flow graph [1, 2]. His work is focuses primarily on timing attacks, while our approach is more general. There are also languages such as Jif and Flowcaml that include information flow support as part of the language [44, 42].

Micali and Reyzin examine "physically observable cryptography" through a framework that is closely related to ours. Their model specifies a "leakage function" (analogous to our notion of transcript) and allows the adversary to measure the outputs of this leakage function on a "physical machine" which may be executing some number of "virtual Turing Machines." Our model, in contrast, is simpler since we consider only a single program executing at a time. Also, Micali and Reyzin focus more on how side channel attacks affect basic theorems of cryptography, while we are interested in automatic transforms that improve security against such attacks [34]. Goldreich and Ostrovsky also proposed a transformation for RAM machines that makes the sequence of memory accesses independent of the input with polylogarithmic slowdown; this would provide security in our sense with a transcript containing such memory accesses, but we are not aware of any practical implementation [17].

The above defenses focus on software; there are also promising solutions that focus on hardware [41, 3, 4, 38]. To coordinate these defenses, we need a contract between hardware researchers and software researchers as to who will protect what. Our transcript is exactly this: a contract specifying what information the software can expect the hardware to leak to the adversary.

7 Conclusion and Open Problems

We presented a program counter model for reasoning about side channel attacks, a system that transforms code to increase resistance against attacks, and a static verifier that checks the code output by our compiler is PC-secure. This framework allows us to prove transformed code is secure against an

interesting class of side channels. With enough work, an even more efficient automatic transformation for PC-security may be possible.

Looking forward, it is an interesting open problem to extend these methods to handle a larger class of side-channel attacks. We have argued that specifying a transcript model as part of the hardware/software interface simplifies development of both hardware and software countermeasures. We leave it as an open problem to find the "right" contract between these two worlds.

8 Acknowledgments

We thank Nikita Borisov, Eric Brewer, Karl Chen, Evan Chang, Adam Chlipala, Rob Johnson, Chris Karlof, Naveen Sastry, Rusty Sears, Umesh Shankar, and Fran Woodland for discussions and advice. We also thank Eric Brewer for reminding us that every problem can be solved with a supercompiler. David Molnar was supported by an Intel OCR Fellowship and a National Science Foundation Graduate Fellowship. This work supported by NSF ANI-0113941 and NSF CCR-0325311.

References

1. Johan Agat. Transforming Out Timing Leaks. In *Proceedings on the 27th ACM Symposium on the Principles of Programming Languages*, 2000.
2. Johan Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Chalmers University of Technology, 2001.
3. Luca Benini, Alberto Macii, Enrico Macii, Elvira Omerbegovic, Massimo Poncino, and Fabrizio Pro. A Novel Architecture for Power Maskable Arithmetic Units. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI*, 2003.
4. Luca Benini, Alberto Macii, Enrico Macii, Elvira Omerbegovic, Massimo Poncino, and Fabrizio Pro. Energy-aware Design Techniques for Differential Power Analysis Protection. In *Proceedings of the 40th conference on Design automation*, 2003.
5. D.J. Bernstein. Cache-timing attacks on AES, 2005. `http://cr.yp.to/papers.html#cachetiming`.
6. John Black and Hector Urtubia. Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
7. D. Bleichenbacher. Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1. In *CRYPTO*, 1998.
8. Johannes Blömer, Jorge Guajardo Merchan, and Volker Krummel. Provably secure masking of AES. In *SAC*, 2004.
9. Johannes Blömer and Jean-Pierre Seifert. Fault based cryptanalysis of the advanced encryption standard (AES). In *FSE 2003*, volume 2742 of *LNCS*, pages 162–181. Springer-Verlag, 2003.
10. Dan Boneh and David Brumley. Remote Timing Attacks Are Practical. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
11. Suresh Chari, Charanjit Jutla, Josyula R. Rao, and Pankaj Rohatgi. A Cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards. In *Proceedings of the Second AES Candidate Conference*, 1999.
12. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, 1999.
13. Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Usenix Security*, 2004.
14. Christophe Clavier and Marc Joye. Universal exponentiation algorithm: A first step towards provable SPA-resistance. In *CHES 2001*, volume 2162 of *LNCS*, pages 300–308. Springer-Verlag, 2001.
15. Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *CHES 1999*, volume 1717 of *LNCS*, pages 292–302. Springer-Verlag, 1999.
16. Jean-Sébastien Coron and Louis Goubin. On Boolean and Arithmetic Masking Against Differential Power Analysis. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, 2000.
17. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAM. *J. ACM*, 43(3):431–473, 1996.
18. H. Handschuh and H. Heys. A timing attack on RC5. In *Lecture Notes in Computer Science: Selected Areas in Cryptography*, pages 306–318. Springer-Verlag, 1999.
19. Matthew Hennessy. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, New York, N.Y., 1990.

20. Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO 2003*, 2003.
21. G. Kahn. Natural semantics. In *STACS*, pages 22–39, Passau, Germany, 1987.
22. John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, 8:141–158, 2000.
23. Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX:5–38, Jan 1883.
24. Vlastimil Klima, Ondrej Pokorny, and Tomas Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, 2003.
25. Vlastimil Klima and Tomas Rosa. Side channel attacks on CBC encrypted messages in the PKCS #7 format. Cryptology ePrint Archive, Report 2003/098, 2003. `http://eprint.iacr.org/`.
26. Paul Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference*, 1996.
27. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference*, 1999.
28. François Koeune and Jean-Jacques Quisquater. A timing attack against Rijndael. Technical Report CG-1999/1, Université catholique de Louvain, June 1999.
29. Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.
30. J. Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In *CRYPTO*, 2001.
31. Thomas S. Messerges. Securing the AES Finalists Against Power Analysis Attacks. In *Proceedings of the Fast Software Encryption Workshop*, 2000.
32. Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Investigations of Power Analysis Attacks on Smartcards. In *Proceedings of the USENIX Workshop on Smartcard Technology*, 1999.
33. Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power Analysis Attacks of Modular Exponentiation in Smartcards. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, 1999.
34. Silvio Micali and Leo Reyzin. Physically observable cryptography. In *Theory of Cryptography*, 2004.
35. Bodo Möller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures, May 2004. `http://www.openssl.org/~bodo/tls-cbc.txt`.
36. George Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the Conference on Compilier Construction*, 2002.
37. D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, University of Bristol, June 2002.
38. Patrick Rakers, Larry Connell, Tim Collins, and Dan Russell. Secure Contactless Smartcard ASIC with DPA Protection. In *Proceedings of the Custom Integrated Circuits Conference*, 2000.
39. Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
40. Fumihiko Sano, Masanobu Koike, Shinichi Kawamura, and Masue Shiba. Performance evaluation of AES finalists on the high-end smart card. In *AES3: The Third Advanced Encryption Standard (AES) Candidate Conference*, Apr 2000.
41. Adi Shamir. Protecting Smart Cards from Passive Power Analysis with Detached Power Supplies. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, 2000.
42. Vincent Simonet. Flowcaml, 2005. `http://cristal.inria.fr/~simonet/soft/flowcaml/`.
43. S. Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS... In *EUROCRYPT*, 2002.
44. Lantian Zheng and Andrew Myers. End-to-end availability policies and noninterference, 2005. Computer Security Foundations Workshop.

A The Transformation

We give a formal definition of our transformation and show that it preserves the semantics of the original program. Although our actual implementation transforms C code, our formal description translates programs written in the simple imperative language called IncredibL, which is based loosely on WhileL [19]. The abstract syntax of this language is given in Figure 7.

A.1 A Natural Semantics for IncredibL

Semantic Domains:
$g \in \text{BreakGuard} = \{eval, skip\}$
$\sigma \in \text{Store} = \text{Identifier} \mapsto (\text{Num} \cup \{\texttt{unbound}\})$

Expression Transition Relations:

$\Longrightarrow_A \quad : \text{Exp} \times \text{Store} \longrightarrow \text{Num}$

Rule NumR
$$\frac{}{\langle n, \sigma\rangle \implies_A n}$$

Rule IdentR
$$\frac{}{\langle I, \sigma\rangle \implies_A \sigma(I)}$$

Rule AOpR
$$\frac{\langle E, \sigma\rangle \implies_A n \quad \langle E', \sigma\rangle \implies_A n'}{\langle E' \text{ arithop } E', \sigma\rangle \implies_A \mathit{apply}(\text{arithop}, n, n')}$$

Rule BitNotR
$$\frac{\langle E, \sigma\rangle \implies_A n}{\langle \tilde{} E, \sigma\rangle \implies_A \mathit{bitwise\text{-}negation}(n)}$$

$\Longrightarrow_B \quad : \text{BoolExp} \times \text{Store} \longrightarrow \{\texttt{0}, \texttt{1}\}$

Rule BOpR
$$\frac{\langle B, \sigma\rangle \implies_B n \quad \langle B', \sigma\rangle \implies_B n'}{\langle B \text{ boolop } B', \sigma\rangle \implies_B \mathit{apply}(\text{boolop}, n, n')}$$

Rule ROpR
$$\frac{\langle E, \sigma\rangle \implies_A n \quad \langle E', \sigma\rangle \implies_A n'}{\langle E \text{ relop } E', \sigma\rangle \implies_B \mathit{apply}(\text{relop}, n, n')}$$

Rule LNotR
$$\frac{\langle B, \sigma\rangle \implies_B n}{\langle ! B, \sigma\rangle \implies_B 1 - n}$$

Fig. 11. Operational Semantics of IncredibL for Arithmetic and Boolean Expressions

The IncredibL language has only three control flow constructs: `if...then...else`, a Pascal-style `for` statement, and a `break` statement to escape from a loop. Loop bounds are required to be constant, which guarantees that IncredibL programs terminate. Although this language is not universal, it is sufficient to implement many cryptographic algorithms, including AES, IDEA, and RC5. Furthermore, we will prove that all programs that can be written in IncredibL can be transformed into observationally equivalent programs that are PC-secure.

Command Transition Relation:

$\Longrightarrow_C$: Com × Store × BreakGuard ⟶ Store × BreakGuard

Rule AsR

$$\frac{\langle E, \sigma\rangle \Longrightarrow_A n}{\langle I \texttt{ := } E, \sigma, eval\rangle \Longrightarrow_C \langle \sigma[n/I], eval\rangle}$$

Rule ComR

$$\frac{\langle C, \sigma, eval\rangle \Longrightarrow_C \langle \sigma', g'\rangle \quad \langle C', \sigma', g'\rangle \Longrightarrow_C \langle \sigma'', g''\rangle}{\langle C; C', \sigma, eval\rangle \Longrightarrow_C \langle \sigma'', g''\rangle}$$

Rule IfR1

$$\frac{\langle B, \sigma\rangle \Longrightarrow_B \texttt{1} \quad \langle C, \sigma, eval\rangle \Longrightarrow_C \langle \sigma', g'\rangle}{\langle \texttt{if } B \texttt{ then } C \texttt{ else } C', \sigma, eval\rangle \Longrightarrow_C \langle \sigma', g'\rangle}$$

Rule IfR2

$$\frac{\langle B, \sigma\rangle \Longrightarrow_B \texttt{0} \quad \langle C', \sigma, eval\rangle \Longrightarrow_C \langle \sigma', g'\rangle}{\langle \texttt{if } B \texttt{ then } C \texttt{ else } C', \sigma, eval\rangle \Longrightarrow_C \langle \sigma', g'\rangle}$$

Rule ForR1

$$\frac{\langle C; \texttt{for } I \texttt{ := } n{+}1 \texttt{ to } n' \texttt{ do } C, \sigma[n/I], eval\rangle \Longrightarrow_C \langle \sigma', g'\rangle}{\langle \texttt{for } I \texttt{ := } n \texttt{ to } n' \texttt{ do } C, \sigma, eval\rangle \Longrightarrow_C \langle \sigma'[\sigma(I)/I], eval\rangle} \quad \text{where } n \leq n'$$

Rule ForR2

$$\frac{}{\langle \texttt{for } I \texttt{ := } n \texttt{ to } n' \texttt{ do } C, \sigma, eval\rangle \Longrightarrow_C \langle \sigma, eval\rangle} \quad \text{where } n > n'$$

Rule BreakR

$$\frac{}{\langle \texttt{break}, \sigma, eval\rangle \Longrightarrow_C \langle \sigma, skip\rangle}$$

Rule SkipR

$$\frac{}{\langle C, \sigma, skip\rangle \Longrightarrow_C \langle \sigma, skip\rangle}$$

Fig. 12. Operational Semantics of IncredibL for Commands

In order to make the transform more concrete, our definition includes all of the bit manipulation necessary to implement it on a 2's complement machine. Therefore, in addition to the usual definitions for addition and subtraction, we assume the following identities for bitwise operators:

$$\begin{aligned}&\forall a : \\ &\quad a \,\&\, (-1) = a \\ &\quad a \,\&\, 0 = 0 \\ &\quad a \,|\, 0 = a \\ &\quad {\sim}(-1) = 0 \\ &\quad {\sim}0 = (-1)\end{aligned}$$

We use σ to denote a *store*, which can be thought of as a piece of memory where variables are stored. The notation $\sigma[n/I]$ represents a new store which is the same as σ except the variable named by I now has the value n. We denote the value of I in store σ as $s(I)$. If I is not bound in the σ, then $s(I)$ has the special value `unbound`.

The natural semantics[21] of IncredibL are given in Figures 11 and 12. As the IncredibL grammar presented in Figure 7 demonstrates, programs consist of commands, which may modify variables and induce control flow. Commands, in turn, may be composed of arithmetic and boolean expressions, which represent mappings from stores to numerical values, but do not have side-effects. We define three transition relations, $\Longrightarrow_A$, $\Longrightarrow_B$, and $\Longrightarrow_C$, for arithmetic expressions, boolean expressions, and commands, respectively. Note that the image of $\Longrightarrow_B$ is a subset of the range of $\Longrightarrow_A$, so boolean expressions are allowed in arithmetic context. This feature is crucial to the implementation of our transformation.

Our operational description is a natural way to express a simple imperative language such as IncredibL. Unfortunately, it does not lend itself well to modelling non-local exit constructs such as `break`, as would a denotational definition. To support `break`, we add a new component called a *`break` guard* to the command configuration. The guard is normally the symbol *eval*, but it is changed to *skip* when a `break` is encountered. In configurations where it has the latter value, the corresponding command is always ignored, and when the first command in a sequence produces a *skip*, the *skip* is propagated through the rest of the sequence. The only command that ignores a *skip* produced by one of its subcomponents is `for`.

Although we omitted the mechanics of it from the grammar of Figure 7 for simplicity, there is also a restriction that well-formed IncredibL programs do not contain any `break` statements outside of a `for` loop. Therefore, the final state of any such program is always of the form $\langle \sigma,\ eval \rangle$, for some store σ.

Subtleties Involving Errors, Loops, and Pointers The careful reader may have noticed that syntactically well-formed programs may exhibit two types of runtime errors: arithmetic errors, such as divide-by-zero and overflow, and uninitialized variable errors. In general, we consider programs that would crash on certain inputs to be beyond the scope of our work. Note, however, that for both types of errors possible in IncredibL, the C language specifies that execution should continue, with the value of the offending expression being undefined. We could adopt similar semantics by asserting that `unbound` evaluates to `0`, for instance. By doing so, we can transform correct and erroneous programs alike, and prove that the resulting code is PC-secure. In practice, it would be better to flag the occurrence of an error and take appropriate action when execution of the algorithm has completed.

IncredibL lacks pointers, which pose a problem in practice for two reasons. First, they introduce data-dependent memory reference patterns and cache behavior. A cache hit has different timing from a cache miss by design, potentially leaking sensitive information. Specific defenses against this type of

side channel are beyond the scope of this paper, and are considered in [37]. Second, in many languages, dereferencing a pointer may lead to side-effects, depending on the value of the pointer. Consider the following snippet of C code, for instance.

```
if (p != NULL)
        *p = 42;
```

If our transformation were applied to this program, the resulting code would construct a mask `m` based on the value of the boolean expression `p != NULL`, then perform an assignment tantamount to `*p = (42 & m) | (*p & ~m)`. Since `NULL` is not a valid memory location, the new program would likely generate a runtime error in cases where `p` is `NULL`, thereby leaking information about `p`.

Another subtle aspect of IncredibL is the particular semantics of loop counter variables. Like Pascal and unlike C, it is not possible to alter the number of iterations the loop will take by assigning directly to the loop counter. Any such assignment will simply be ignored on the next loop iteration. This is necessary to ensure that all IncredibL programs terminate.

Furthermore, unlike other variables, loop counters are in scope only for the duration of the loop. The operational semantics encode this property by restoring the original value (even if that value is `unbound`) at the end of every loop iteration. There is no fundamental reason to impose this restriction; however, it simplifies the transform significantly. In particular, with these semantics, it is unnecessary for the transform to nullify the changes to the induction variable in cases where the loop would not have been executed in the original program. It also allows us to avoid the issue of how to ensure that the counter has the correct value if the original program uses a `break` to escape from the loop.

A.2 Transformation

The transform in Figure 8 is specified in terms of three functions:

$$\mathcal{T}_{\text{Program}} : \text{Program} \mapsto \text{Program}$$
$$\mathcal{T}_{\text{Com}} : \text{Com} \times \text{Identifier} \times \text{Identifier} \mapsto \text{Com}$$
$$\textit{conditional-assign} : \text{Identifier} \times \text{Exp} \times \text{Exp} \times \text{Exp} \mapsto \text{Com}$$

$\mathcal{T}_{\text{Com}}$ does most of the work, and it can be thought of as an internal subroutine to $\mathcal{T}_{\text{Program}}$. The identifiers passed as arguments to $\mathcal{T}_{\text{Com}}$ name variables that will store the control context of the corresponding expression at runtime (in the form of masks). Com is a union type, which may represent an assignment command; a sequence of commands; or an `if`, `for`, or `break` command. Therefore, we deconstruct the first argument to $\mathcal{T}_{\text{Com}}$ using pattern matching, much as one would do in ML. For example, the notation $\mathcal{T}_{\text{Com}}[\![I \texttt{ := } E]\!](I_{\text{if}}, I_{\text{brk}}) = \cdots$ indicates that if the command argument is of the form I `:=` E, then the corresponding rule should be applied with I and E bound appropriately. I_{if} and I_{brk}, on the other hand, are ordinary formal parameters.

The *conditional-assign* function implements an abstraction of conditional assignment, without using `if`. That is, *conditional-assign*(I, E_m, E_t, E_f) produces a command that is equivalent to I `:=` E_t if E_m evaluates to `-1` at runtime, and equivalent to I `:=` E_f if E_m evaluates to `0`. Our transform enforces the property that *conditional-assign* is never called with an E_m that may evaluate to something other than `0` or `-1`. We prove in Lemma 1 that our implementation of *conditional-assign* behaves as advertised. However, *conditional-assign* is merely an abstraction, and it may have a more efficient implementation on particular architectures, e.g. using a conditional `mov` instruction. Admittedly, we neglected to develop an abstraction for the representation of the masks, but doing so would not be difficult.

We see that end result of our transform is a new program with no data-dependent control flow. All `if` statements are replaced by blocks of conditional assignments and `break` statements in for loops replaced by assignments. Therefore the output of the transformation contains no `if` or `break` statements.

Combined with the fact IncredibL only allows loops with a fixed constant number of iterations, we can take the above discussion as a proof sketch for the following theorem:

Theorem 5. *For every IncredibL program P, $\mathcal{T}_{Program}[\![P]\!]$ consists only of straight-line code and (possibly nested) loops with straight-line bodies that run for a fixed constant number of iterations with no assignments to induction variables.*

Recall that the bounds of a `for` loop are lexical constants. Since the transformation is able to eliminate `break` statements, it would even be possible to fully unroll all loops statically, producing straight-line code. In fact, the transition rule **ForR1** effectively performs this unrolling. Therefore, we can view Theorem 5 as a simple extension of Theorem 1. We do not actually perform the unrolling as part of the transform, though, because doing so results in a worst-case exponential blowup in program size.

A.3 Proof of Semantic Preservation

The transformation introduces new *fresh* identifiers to model the control state of the original program. These identifers are guaranteed not to appear in the original code, so their introduction does not affect the meaning of the program. When we wish to differentiate these two types of identifiers, we will write the store as the union of σ, which contains the variables that were present in the original program, and $\sigma_{\mathcal{T}}$, which contains variables added by the transformation. By definition, σ and $\sigma_{\mathcal{T}}$ are disjoint.

Definition 3 *$C \sim_{\sigma_{\mathcal{T}}} C'$ if, for all stores σ,*

$$\langle C,\ \sigma,\ eval\rangle \Longrightarrow_C \langle \sigma',\ eval\rangle$$

if and only if

$$\langle C',\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma' \cup \sigma'_{\mathcal{T}},\ eval\rangle.$$

Remark 3. Intuitively, one might try to say that two commands C and C' are observationally equivalent if, for all input stores, they produce the same output store. However, to allow us to prove that the transformation is semantically preserving, the definition above excludes any new variables used internally by the transform.

Lemma 1. ***Correctness of conditional-assign.*** *If $\sigma_{\mathcal{T}}(E_m) = -1$, then*

$$(I\ \texttt{:=}\ E_t) \sim_{\sigma_{\mathcal{T}}} \textit{conditional-assign}(I, E_m, E_t, E_f).$$

If $\sigma_{\mathcal{T}}(E_m) = 0$, then

$$(I\ \texttt{:=}\ E_f) \sim_{\sigma_{\mathcal{T}}} \textit{conditional-assign}(I, E_m, E_t, E_f).$$

Proof. First, note that $\textit{conditional-assign}(I, E_m, E_t, E_f) = (I\ \texttt{:=}\ \texttt{(}E_t\ \texttt{\&}\ E_m\texttt{)}\ \texttt{|}\ \texttt{(}E_f\ \texttt{\&}\ \texttt{\~{}}E_m\texttt{)})$. Observe that by the transition rules for assignment and expressions, we can replace E_m with its value in $\sigma_{\mathcal{T}}$ to obtain a command that is equivalent with respect to $\sigma_{\mathcal{T}}$. If $\sigma_{\mathcal{T}}(E_m) = -1$, then by the identities for bitwise operators described previously,

$$\begin{aligned} I\ \texttt{:=}\ \texttt{(}E_t\ \texttt{\&}\ \texttt{-1)}\ \texttt{|}\ \texttt{(}E_f\ \texttt{\&}\ \texttt{\~{}-1)} &\sim_{\sigma_{\mathcal{T}}} I\ \texttt{:=}\ E_t\ \texttt{|}\ \texttt{(}E_f\ \texttt{\&}\ \texttt{0)} \\ &\sim_{\sigma_{\mathcal{T}}} I\ \texttt{:=}\ E_t\ \texttt{|}\ \texttt{0} \\ &\sim_{\sigma_{\mathcal{T}}} I\ \texttt{:=}\ E_t \end{aligned}$$

The proof for the $E_m = 0$ case proceeds analogously.

The following lemma states that $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}})$ is essentially a no-op with respect to any store where either I_{if} or I_{brk} is 0.

Lemma 2. *For all commands C, if $\sigma_{\mathcal{T}}(I_{if}) = 0$ or $\sigma_{\mathcal{T}}(I_{brk}) = 0$, then*

$$\langle \mathcal{T}_{Com}[\![C]\!](I_{if}, I_{brk}),\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma \cup \sigma'_{\mathcal{T}},\ eval\rangle.$$

Furthermore, $\sigma_{\mathcal{T}}(I_{if}) = \sigma'_{\mathcal{T}}(I_{if})$ and $\sigma_{\mathcal{T}}(I_{brk}) = \sigma'_{\mathcal{T}}(I_{brk})$.

Proof. Consider all five possible deconstructions of C:

1. If $C = I$ `:=` E, then $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}}) = \textit{conditional-assign}(I, \texttt{(}I_{\text{if}}\ \texttt{\&}\ I_{\text{brk}}\texttt{)}, E, I)$. The expression `(`$I_{\text{if}}$ `&` I_{brk}`)` evaluates to `0` with respect to $\sigma_{\mathcal{T}}$, so by Lemma 1, the conditional assignment is equivalent to I `:=` I. Transition rules **AsR** and **IdentR** yield

   $$\langle I \texttt{ := } I,\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C^* \langle \sigma[\sigma(I)/I] \cup \sigma_{\mathcal{T}},\ eval\rangle$$

   Observe that $\sigma[\sigma(I)/I] = \sigma$.
2. If $C = C'$; C'', then $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}}) = \mathcal{T}_{\text{Com}}[\![C']\!](I_{\text{if}}, I_{\text{brk}})$; $\mathcal{T}_{\text{Com}}[\![C'']\!](I_{\text{if}}, I_{\text{brk}})$. We may assume by structural induction that the following statements hold:

   $$\langle \mathcal{T}_{\text{Com}}[\![C']\!](I_{\text{if}}, I_{\text{brk}}),\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma \cup \sigma'_{\mathcal{T}},\ eval\rangle$$
   $$\sigma_{\mathcal{T}}(I_{\text{if}}) = \sigma'_{\mathcal{T}}(I_{\text{if}}) \text{ and } \sigma_{\mathcal{T}}(I_{\text{brk}}) = \sigma'_{\mathcal{T}}(I_{\text{brk}})$$

   The second line implies that one of $\sigma'_{\mathcal{T}}(I_{\text{if}})$ and $\sigma'_{\mathcal{T}}(I_{\text{brk}})$ is `0`, so we may also conclude by induction that

   $$\langle \mathcal{T}_{\text{Com}}[\![C'']\!](I_{\text{if}}, I_{\text{brk}}),\ \sigma \cup \sigma'_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma \cup \sigma''_{\mathcal{T}},\ eval\rangle.$$
   $$\sigma_{\mathcal{T}}(I_{\text{if}}) = \sigma'_{\mathcal{T}}(I_{\text{if}}) = \sigma''_{\mathcal{T}}(I_{\text{if}}) \text{ and } \sigma_{\mathcal{T}}(I_{\text{brk}}) = \sigma'_{\mathcal{T}}(I_{\text{brk}}) = \sigma''_{\mathcal{T}}(I_{\text{brk}})$$

   Then by transition rule **ComR**, $\langle \mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}}),\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma \cup \sigma''_{\mathcal{T}},\ eval\rangle$.
3. If $C =$ `if` B `then` C' `else` C'', then $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}})$ introduces two conditional assignments of the form $\textit{conditional-assign}(I_0, \texttt{(}I_{\text{if}}\ \texttt{\&}\ I_{\text{brk}}\texttt{)}, \square, \texttt{0})$, where I_0 is a new identifier. Since `(`I_{if} `&` I_{brk}`)` evaluates to `0` with respect to $\sigma_{\mathcal{T}}$, Lemma 1 implies that both of these conditional assignments are equivalent to I_0 `:=` `0`. Therefore, by induction:

   $$\langle \mathcal{T}_{\text{Com}}[\![C']\!](I_0, I_{\text{brk}}),\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma \cup \sigma'_{\mathcal{T}},\ eval\rangle$$
   $$\langle \mathcal{T}_{\text{Com}}[\![C'']\!](I_0, I_{\text{brk}}),\ \sigma \cup \sigma'_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma \cup \sigma''_{\mathcal{T}},\ eval\rangle$$

   Hence, $\langle \mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}}),\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma \cup \sigma''_{\mathcal{T}}[\texttt{0}/I_0],\ eval\rangle$. We also note that $\sigma_{\mathcal{T}}(I_{\text{if}}) = \sigma''_{\mathcal{T}}(I_{\text{if}})$ and $\sigma_{\mathcal{T}}(I_{\text{brk}}) = \sigma''_{\mathcal{T}}(I_{\text{brk}})$ for the same reasons as in case 2.
4. Suppose $C =$ `for` I `:=` n `to` n' `do` C'. Just as in case 3, the conditional assignment in the transform of the `for` is reducible to I_0 `:=` `0`. Hence, the evaluation of $\mathcal{T}_{\text{Com}}[\![C']\!](I_{\text{if}}, I_0)$ preserves σ and $\sigma_{\mathcal{T}}(I_{\text{if}})$ by structural induction. If $n > n'$, then rule **ForR2** applies and the proof is trivial. Otherwise, rule **ForR1** applies and we assume by induction on n that

   $$\langle \texttt{for } I \texttt{ := } n+1 \texttt{ to } n' \texttt{ do } \mathcal{T}_{\text{Com}}[\![C']\!](I_{\text{if}}, I_0),\ \sigma[n/I] \cup \sigma'_{\mathcal{T}},\ eval\rangle \ \Longrightarrow_C \ \langle \sigma[n/I] \cup \sigma''_{\mathcal{T}},\ eval\rangle$$

   where $\sigma''_{\mathcal{T}}(I_{\text{if}}) = \sigma'_{\mathcal{T}}(I_{\text{if}}) = \sigma_{\mathcal{T}}(I_{\text{if}})$. Therefore, the conclusion of **ForR1** asserts that the entire `for` expression evaluates to $\langle \sigma[n/I][\sigma(I)/I] \cup \sigma''_{\mathcal{T}},\ eval\rangle = \langle \sigma \cup \sigma''_{\mathcal{T}},\ eval\rangle$.
5. If $C =$ `break`, then $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}}) = \textit{conditional-assign}(I_{\text{brk}}, \texttt{(}I_{\text{if}}\ \texttt{\&}\ I_{\text{brk}}\texttt{)}, \texttt{0}, I_{\text{brk}})$. By assumption, either I_{if} or I_{brk} is `0` in $\sigma_{\mathcal{T}}$, so by Lemma 1, the conditional assignment is equivalent to I_{brk} `:=` I_{brk}. Therefore, the store is unmodified. □

Now that we have proven that $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}})$ is a no-op when either I_{if} or I_{brk} is 0, we can show that the same command is almost equivalent to C with respect to a store where I_{if} and I_{brk} are both -1. We say "almost" because the transform's behavior differs in two important ways. First, it adds new variables to the store, as discussed previously in this section. Second, the transform will not contain any `break` commands, so it will never produce a BreakGuard other than *eval*. However, where the original program would have executed a `break`, the transform will set I_{brk} to 0. Lemma 2 implies that this has the effect of causing the transform to nullify all subsequent effects of the loop.

Lemma 3. *For any command C, if $\sigma_{\mathcal{T}}(I_{if}) = \sigma_{\mathcal{T}}(I_{brk}) = -1$, then*

$$(\langle C,\ \sigma,\ eval\rangle \Longrightarrow_C \langle \sigma',\ g\rangle) \quad \Longleftrightarrow \quad (\langle \mathcal{T}_{Com}[\![C]\!](I_{if}, I_{brk}),\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma' \cup \sigma'_{\mathcal{T}},\ eval\rangle)$$

Moreover, $\sigma'_{\mathcal{T}}(I_{if}) = -1$ and

$$\sigma'_{\mathcal{T}}(I_{brk}) = \begin{cases} -1, & \text{if } g = eval \\ 0, & \text{if } g = skip \end{cases}$$

Proof. Once again we analyze how the transform behaves for each kind of command.

1. If $C = I$ `:=` E, then $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}}) = \textit{conditional-assign}(I, (I_{\text{if}}\ \texttt{\&}\ I_{\text{brk}}), E, I)$, where `(`$I_{\text{if}}$ `&` I_{brk}`)` evaluates to `-1`. Therefore by Lemma 1, $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}}) \sim_{\sigma_{\mathcal{T}}} C$, which clearly implies the desired result.
2. If $C = C'$; C'', then $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}}) = \mathcal{T}_{\text{Com}}[\![C']\!](I_{\text{if}}, I_{\text{brk}})$; $\mathcal{T}_{\text{Com}}[\![C'']\!](I_{\text{if}}, I_{\text{brk}})$. Suppose that $\langle C',\ \sigma,\ eval\rangle \Longrightarrow_C \langle \sigma',\ g'\rangle$ and $\langle C'',\ \sigma',\ eval\rangle \Longrightarrow_C \langle \sigma'',\ g''\rangle$. Then by rule **ComR**, we conclude $\langle C,\ \sigma,\ eval\rangle \Longrightarrow_C \langle \sigma'',\ g''\rangle$. We assume by structural induction that

   $$\langle \mathcal{T}_{\text{Com}}[\![C']\!](I_{\text{if}}, I_{\text{brk}}),\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma' \cup \sigma'_{\mathcal{T}},\ eval\rangle$$
   $$\sigma'_{\mathcal{T}}(I_{\text{if}}) = -1, \text{ and } \sigma'_{\mathcal{T}}(I_{\text{brk}}) = -1 \text{ if } g' = eval \text{ and } 0 \text{ otherwise}$$

   There are now two subcases:
   - If $g' = skip$, then $\sigma' = \sigma''$. Moreover, $\sigma'_{\mathcal{T}}(I_{\text{brk}}) = 0$, so by Lemma 2,

     $$\langle \mathcal{T}_{\text{Com}}[\![C'']\!],\ \sigma' \cup \sigma'_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma' \cup \sigma''_{\mathcal{T}},\ eval\rangle$$

     and $\sigma''_{\mathcal{T}}(I_{\text{if}}) = \sigma'_{\mathcal{T}}(I_{\text{if}})$ and $\sigma''_{\mathcal{T}}(I_{\text{brk}}) = \sigma'_{\mathcal{T}}(I_{\text{brk}}) = 0$.
   - If $g' = eval$, then $\sigma'_{\mathcal{T}}(I_{\text{brk}}) = -1$. Hence we assume by induction that

     $$\langle \mathcal{T}_{\text{Com}}[\![C'']\!],\ \sigma' \cup \sigma'_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma'' \cup \sigma''_{\mathcal{T}},\ eval\rangle$$

     and that the usual invariants hold for $\sigma''_{\mathcal{T}}(I_{\text{if}})$ and $\sigma''_{\mathcal{T}}(I_{\text{brk}})$.
3. Suppose $C =$ `if` B `then` C' `else` C''. The conditional assignments ensure that I_0 has value 0 for the branch that would be evaluated in the untransformed program, and -1 for the other branch. If the consequent is taken, then the theorem holds for $\mathcal{T}_{\text{Com}}[\![C']\!](I_0, I_{\text{brk}})$ by induction. The remaining code in the sequence, corresponding to $\mathcal{T}_{\text{Com}}[\![C'']\!](I_0, I_{\text{brk}})$, is effectively a no-op by Lemma 2, so it does not affect the invariants we seek to prove. Similarly, if the alternative is taken, then the transformed consequent is a no-op, and $\mathcal{T}_{\text{Com}}[\![C'']\!](I_0, I_{\text{brk}})$ behaves as desired by induction.
4. Suppose $C =$ `for` I `:=` n `to` n' `do` C'. Note that the conditional assignment introduced in the transform $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}})$ initializes I_0 to -1. If $n > n'$ or $\langle C,\ \sigma,\ eval\rangle \Longrightarrow_C \langle \sigma',\ skip\rangle$, then the transformed loop is clearly equivalent to the original one. Else note that for some σ', σ'',

   $$\langle C',\ \sigma,\ eval\rangle \Longrightarrow_C \langle \sigma',\ eval\rangle$$
   $$\langle \texttt{for}\ I\ \texttt{:=}\ n+1\ \texttt{to}\ n'\ \texttt{do}\ C',\ \sigma',\ eval\rangle \Longrightarrow_C \langle \sigma'',\ eval\rangle.$$

By structural induction, $\langle \mathcal{T}_{\text{Com}}[\![C']\!](I_{\text{if}}, I_0),\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma' \cup \sigma'_{\mathcal{T}},\ eval\rangle$ where $\sigma'_{\mathcal{T}}(I_{\text{if}}) = -1$ and $\sigma'_{\mathcal{T}}(I_{\text{brk}}) = -1$. By application of rule **ComR** and induction on n:

$$\langle \texttt{for}\ I\ \texttt{:=}\ n+1\ \texttt{to}\ n'\ \texttt{do}\ \mathcal{T}_{\text{Com}}[\![C']\!](I_{\text{if}}, I_0),\ \sigma' \cup \sigma'_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma'' \cup \sigma''_{\mathcal{T}},\ eval\rangle$$

where $\sigma''_{\mathcal{T}}(I_{\text{if}}) = -1$. The command $\mathcal{T}_{\text{Com}}[\![C']\!](I_{\text{if}}, I_0)$ is unaware of I_{brk}, since it is passed I_0 instead, so $\sigma''_{\mathcal{T}}(I_{\text{brk}}) = \sigma'_{\mathcal{T}}(I_{\text{brk}}) = -1$.

5. If $C = \texttt{break}$, then $\langle C,\ \sigma,\ eval\rangle \Longrightarrow_C \langle \sigma,\ skip\rangle$ by rule **BreakR**. By definition, $\mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}}) = \textit{conditional-assign}(I_{\text{brk}}, (I_{\text{if}}\ \texttt{\&}\ I_{\text{brk}}), \texttt{0}, I_{\text{brk}})$, and $(I_{\text{if}}\ \texttt{\&}\ I_{\text{brk}})$ evaluates to -1 with respect to $\sigma_{\mathcal{T}}$. By Lemma 1, this conditional assignment is equivalent to $I_{\text{brk}}\ \texttt{:=}\ \texttt{0}$. Therefore

$$\langle \mathcal{T}_{\text{Com}}[\![C]\!](I_{\text{if}}, I_{\text{brk}}),\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle \Longrightarrow_C \langle \sigma \cup \sigma'_{\mathcal{T}},\ eval\rangle$$

where $\sigma'_{\mathcal{T}}(I_{\text{brk}}) = 0$ and $\sigma'_{\mathcal{T}}(I_{\text{if}}) = \sigma_{\mathcal{T}}(I_{\text{if}}) = -1$. □

Theorem 6. *For all IncredibL programs $P = C$, $C \sim_{\sigma_{\mathcal{T}}} \mathcal{T}_{Program}[\![C]\!]$.*

Proof. For any store σ, $\langle C,\ \sigma,\ eval\rangle \Longrightarrow_C \langle \sigma',\ eval\rangle$ for some σ', since top-level `break`s are not allowed. Let $\sigma_{\mathcal{T}}$ denote an empty store, so $\sigma \cup \sigma_{\mathcal{T}} = \sigma$. We have

$$\begin{aligned}
\langle \mathcal{T}_{\text{Program}}[\![C]\!],\ \sigma,\ eval\rangle &= \langle I_0\ \texttt{:= -1;}\ \mathcal{T}_{\text{Com}}[\![C]\!](I_0, I_0),\ \sigma \cup \sigma_{\mathcal{T}},\ eval\rangle && \text{by definition of } \mathcal{T}_{\text{Program}} \\
&\Longrightarrow_C \langle \mathcal{T}_{\text{Com}}[\![C]\!](I_0, I_0),\ \sigma \cup \sigma_{\mathcal{T}}[\texttt{-1}/I_0],\ eval\rangle && \text{by rules } \textbf{ComR} \text{ and } \textbf{AsR} \\
&\Longrightarrow_C \langle \sigma' \cup \sigma'_{\mathcal{T}},\ eval\rangle && \text{by Lemma 3}
\end{aligned}$$

Therefore, $C \sim_{\sigma_{\mathcal{T}}} \mathcal{T}_{\text{Program}}[\![C]\!]$.

A More Transcript Models

The transcript model can be thought of as a *class* of threat models. This generality makes it possible to state definitions that apply to many types of side channel attacks, as we do in § 3, but we must consider particular *instances* of the transcript model when analyzing real-world systems. Examples of the transcript model instances include:

Program counter model. The adversary obtains a list of the program counters reached by a program during its execution. In a program counter transcript, each step corresponds to a single instruction execution, and T_i is the memory location of the i^{th} instruction executed.

Hamming weight models. Here the transcript consists of a list of Hamming weights of intermediate values in the computation. The particular weights available to the adversary might correspond to the results of each round of a cipher, as in [22], or it could be a hardware-centric notion involving loads and stores or cache behavior ([2, 5]).

Intermediate result models. These models are similar to Hamming weight models, except that the adversary is able to observe a few bits of internal state (possibly at locations selected by the adversary). The transcript is a sequence of such observed values. Several previous papers fit into this model ([8, 20, 12]).

Differential Power Analysis. Differential power analysis uses traces of a device's power consumption taken at a particular sampling rate. We can capture this in the transcript model by specifying a *power sample transcript*. In such a transcript, at each step the adversary receives a single power sample T_i. [4]

Probing attacks. Along similar lines, one can consider probing attacks, where the adversary is able to see the value of a few bits of internal state (possibly at locations selected by the adversary). The transcript is the sequence of such observed values. Several previous papers fit into this model [8, 20, 12].

It is an interesting open problem to try to apply our methods to any of these models, or to develop other models that capture information that might be available to the adversary in a side-channel attack.

[4] To make such a specification complete, we must specify an *energy model*, which tells us how much energy is consumed by an instruction given the execution history, state of the machine, and instruction arguments. One simplified energy model is the Hamming model, where the observed power consumption is correlated to the Hamming weight of various intermediate values ; one can imagine many other energy models as well.