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.

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 [19] 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" [17]. Kocher, Jaffe, and Jun showed how careful power measurements could reveal private keys through "power analysis" [18]. Since then, side channel attacks have been

used to break numerous smart card implementations of both symmetric and public-key cryptography [10, 22, 21, 23]. Later, Boneh and Brumley showed that a timing attack could be mounted against a remote web server [9]. 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 [20, 6, 32, 7, 15, 16, 25].

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.

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 (§ 5.1).

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.

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 [24].

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 $S^{P_k} \sim A^{\#P_k\#}$.*

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

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.

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. 1. 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.

4 Example: Error Disclosure Side Channels

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, 32, 7, 15, 16, 25].

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, 32, 7, 15, 16, 25, 20]. See, for instance, Fig. 1(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. We show several similar applications of PC-security in the full paper [26].

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. 1(b) one possible implementation strategy for securing the code in Fig. 1(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 subrou-

tines 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(false) $= 0$ and MASK(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.

4.1 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.*

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.

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 full paper that this can be done for any fragment of code where all loops executed for a bounded number of iterations [26].

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 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)`.

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 Appendix A

[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.

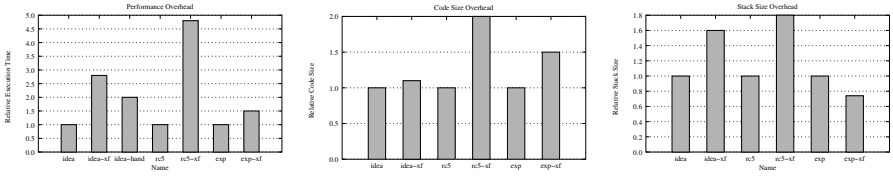

Fig. 2. The speed, code size, and stack size overhead of our transform, as applied to modular exponentiation, IDEA, and RC5. The -xf suffix indicates the automatic application of our transform, while the -hand suffix indicates a hand-optimized application of our transform. Values are normalized: the 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.

5.1 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 [14, 12]. 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 [27]. 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. 2. 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 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. Details of our checker's construction and operation may be found in the full version of the paper [26].

6 Related Work

Many previous side channel defenses are application-specific. For example, blinding can be used to prevent timing attacks against RSA [17, 9]. 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 § 5.1.

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 [21]. 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 [11]. 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 [29], 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 [33, 31].

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 [24].

The above defenses focus on software; there are also promising solutions that focus on hardware [30, 3, 4, 28]. 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. 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. Dan Boneh and David Brumley. Remote Timing Attacks Are Practical. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
10. 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.
11. 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.
12. 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.
13. Matthew Hennessy. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, New York, N.Y., 1990.
14. John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, 8:141–158, 2000.
15. Vlastimil Klima, Ondrej Pokorny, and Tomas Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, 2003.
16. 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/`.
17. Paul Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference*, 1996.
18. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference*, 1999.
19. Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.
20. J. Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In *CRYPTO*, 2001.

21. Thomas S. Messerges. Securing the AES Finalists Against Power Analysis Attacks. In *Proceedings of the Fast Software Encryption Workshop*, 2000.
22. 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.
23. 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.
24. Silvio Micali and Leo Reyzin. Physically observable cryptography. In *Theory of Cryptography*, 2004.
25. Bodo Möller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures, May 2004. `http://www.openssl.org/~bodo/tls-cbc.txt`.
26. David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks (Full Version), 2005. IACR eprint archive report 2005/368.
27. 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.
28. 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.
29. Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
30. 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.
31. Vincent Simonet. Flowcaml, 2005. `http://cristal.inria.fr/~simonet/soft/flowcaml/`.
32. S. Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS... In *EUROCRYPT*, 2002.
33. Lantian Zheng and Andrew Myers. End-to-end availability policies and noninterference, 2005. Computer Security Foundations Workshop.

A Specifying the Transform

We are now ready to specify the transform more precisely. As we discussed, 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 [13]. The grammar for IncredibL can be found in Fig. 3.

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. 4. We state the main theorems here. Proofs can be found in full version [26].

Theorem 3. *$\mathcal{T}_{Program}$ is semantics-preserving: 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.*

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

(a) Syntactic domains.

$$
\begin{array}{l}
C ::= \\
\quad I \texttt{ := } E \mid C'\texttt{; } C'' \mid \texttt{if } B \texttt{ then } C' \texttt{ else } C'' \\
\qquad \mid \texttt{for } I \texttt{ := } n \texttt{ to } n' \texttt{ do } C' \mid \texttt{break} \\
\\
E ::= \quad I \mid n \mid B \mid E' \text{ arithop } E'' \mid \texttt{\~{}}E' \\
\\
B ::= \\
\quad 0 \mid 1 \mid B' \text{ boolop } B'' \mid E' \text{ relop } E'' \mid \texttt{!}B'
\end{array}
$$

(b) Grammar.

Fig. 3. The abstract syntax of IncredibL.

$$
\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}}) \\
&\qquad \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) \\
&\qquad \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. 4. A formal specification of our transform.