Formalized static analysis of constant-time cryptographic algorithms

Gustavo Betarte

Instituto de Computación, Facultad de Ingeniería
Universidad de la República

Jornadas de Ciencia de la Computación

Rosario, October 2014

Joint work with:

Gilles Barthe (IMDEA Software), Juan D. Campo, Carlos Luna (InCo) and David Pichardie (INRIA - ENS Rennes)

Plan

Motivation

Dataflow analysis and the DFP

Language based security

Mitigation of cache-based attacks against crypto-algorithms

Motivation

- Cache-based attacks are a class of side-channel attacks that are particularly effective in virtualized or cloud-based environments
- Countermeasure: to use constant-time implementations, i.e. which do not branch on secrets and do not perform memory accesses that depend on secrets
- There was no rigorous proof that constant-time implementations are protected against concurrent cache-attacks in virtualization platforms with shared cache
- New software mechanism: Stealth memory provisions a small amount of private cache for programs to carry potentially leaking computations securely (S-constant-time).
- No rigorous analysis of stealth memory and S-constant-time, and no tool support for checking if applications are S-constant-time
- To develop a new information-flow analysis that checks if an x86 application executes in constant-time, or in S-constant-time and to prove that constant-time (resp. S-constant-time) programs do not leak confidential information through the cache to other operating systems executing concurrently on virtualization platforms
- To formalize the results using the Coq proof assistant and to demonstrate the effectiveness of our analyses on widely used implementations of cryptographic algorithms

Dataflow analysis

Simple example

- Compilers can perform some optimizations based only on local information

$$\mathtt{x} = \mathtt{a} + \mathtt{b};$$
$$\mathtt{x} = 5 * 2;$$

- The first assignment to `x` is a *useless* assignment: the value computed for `x` is never used
- The expression `5*2` can be computed at compile time, simplifying the second assignment statement to `x = 10`
- Some optimizations require more *global* information

Dataflow analysis

Motivation

```
a = 1;
b = 2;
c = 3;
if (...)  x = a + 5;
else x = b + 4;
c = x + 1;
```

- The initial assignment to `c` (at line 3) is useless, and the expression `x + 1` can be simplified to `7`
- It is less obvious how a compiler can discover these facts
- To discover these kinds of properties it is used dataflow analysis
- Dataflow analysis is usually performed on the program's control-flow graph (CFG)
- The goal is to associate with each program component (each node of the CFG) information that is guaranteed to hold at that point on all executions.

Application of data flow analysis

Constant propagation

- Goal: to determine where in the program variables are guaranteed to have constant values
- More specifically, the information computed for each CFG node n is a set of pairs, each of the form (*variable*, *value*)
- To have the pair (x, v) at node n means that x is guaranteed to have value v whenever n is reached during program execution

Other applications

- Live analysis
- Available expressions
- Reaching definitions
- Common expressions
- (Java) Bytecode verification
- Taint analysis for code injection prevention
- Secure Information flow verification

An informal characterization of (forward) DFP

When we do dataflow analysis "by hand", we look at the *CFG* and think about:

1. What information holds at the start of the program

An informal characterization of (forward) DFP

When we do dataflow analysis "by hand", we look at the *CFG* and think about:

1. What information holds at the start of the program
2. When a node *n* has more than one incoming edge in the *CFG*, how to combine the incoming information (i.e., given the information that holds after each predecessor of *n*, how to combine that information to determine what holds before *n*)

An informal characterization of (forward) DFP

When we do dataflow analysis "by hand", we look at the *CFG* and think about:

1. What information holds at the start of the program
2. When a node *n* has more than one incoming edge in the *CFG*, how to combine the incoming information (i.e., given the information that holds after each predecessor of *n*, how to combine that information to determine what holds before *n*)
3. How the execution of each node changes the information

More formally

An instance of a DFP includes:

- a CFG

More formally

An instance of a DFP includes:

- a CFG
- a domain *D* of *dataflow facts*,

More formally

An instance of a DFP includes:

- a CFG
- a domain *D* of *dataflow facts*,
- a dataflow fact *init* (the information true at the start of the program for forward problems, or at the end of the program for backward problems),

More formally

An instance of a DFP includes:

- a CFG
- a domain D of *dataflow facts*,
- a dataflow fact *init* (the information true at the start of the program for forward problems, or at the end of the program for backward problems),
- an operator $\sqcap$ (used to combine incoming information from multiple predecessors),

More formally

An instance of a DFP includes:

- a CFG
- a domain D of *dataflow facts*,
- a dataflow fact *init* (the information true at the start of the program for forward problems, or at the end of the program for backward problems),
- an operator $\sqcap$ (used to combine incoming information from multiple predecessors),
- for each CFG node n, a dataflow function $f_n : D \to D$ (defines the effect of executing n, also called the transfer function)

Constant propagation as a DFP instance

- $D \;= \wp(X \times V)$
- $init \;= \{\}$
- $\sqcap = \cap$
- if n is not an assignment in CFG, then $f_n(d) = d$, otherwise $(x = e)$
  1. If the right-hand side e has a variable that is not constant, then $f_n(d) = d - (x, *)$
  2. If all right-hand-side variables have constant values, then the right-hand side of the assignment is evaluated producing constant-value c, and $f_n(d) = d - (x, *) \cup \{(x, c)\}$

What is a correct solution of a DFP?

- A solution to an instance of a dataflow problem is a dataflow fact for each node of the given CFG, but
  - what does it mean for a solution to be correct, and
  - if there is more than one correct solution, how can we judge whether one is better than another?
- Ideally, we would like the information at a node to reflect what might happen on all possible paths to that node.
- This ideal solution is called the meet over all paths (MOP) solution
- It is not always possible to compute the MOP solution; we must sometimes settle for a solution that provides less precise information

The MOP solution

The MOP solution (for a forward problem) for each CFG node n is defined as follows:

- For every path $enter \to \ldots \to n$, compute the dataflow fact induced by that path
- Combine the computed facts (using the combining operator, $\sqcap$).
- The result is the MOP solution for node n.

DFP solving using iterative algorithms

Most of the iterative algorithms are variations on the following algorithm (this version is for forward problems):

(Step 1) (initialize n.afters):
Set enter.after = init. Set all other n.after to T.
(Step 2) (initialize worklist):
Initialize a worklist to contain all CFG nodes except enter and exit
(Step 3) (iterate):
While the worklist is not empty:
  Remove a node n from the worklist
  Compute n.before by combining all p.after such that p is a pred. of n in the CFG
  Compute tmp = f_n (n.before)
  If (tmp != n.after) then
    Set n.after = tmp
    Put all of n's successors on the worklist

T (called *top*) has the following properties

- for all dataflow facts d, T $\sqcap$ d = d.
- for all dataflow functions, $f_n(T) = T$.

The Lattice model of data flow analysis

Questions to address

- The definition of DFP includes a domain D of *dataflow facts*, a dataflow fact *init*, an operator $\sqcap$ and for each CFG node n, a dataflow function $f_n : D \to D$
- Goal: to solve a given instance of the problem by computing *before* and *after* sets for each node of the CFG.
- With no additional information about D, $\sqcap$ and f_n, we can't say, in general, whether a particular algorithm for computing the before and after sets works correctly:
  - does the algorithm always halt?
  - does it compute the MOP solution?
  - if not, how does the computed solution relate to the MOP solution?

The Lattice model of data flow analysis

Kildall's framework

- G. Kildall (Kildall 1973) addressed the questions by putting the following additional requirements:
  1. D must be a complete lattice L such that for any instance of the dataflow problem, L has no infinite descending chains
  2. $\sqcap$ must be the lattice's meet operator
  3. f_n must be distributive
  4. the iterative algorithm must initialize n.after (for all nodes n other than the enter node) to the lattice's *top* value
- Given these properties, Kildall showed that:
  - The iterative algorithm always terminates
  - The computed solution is the MOP solution

Language based security

- The goal of language-based security is to provide enforcement mechanisms for end-to-end security policies
- In contrast to security models based on access control, language-based security focuses on information flow policies that track how sensitive information is propagated during execution.
- Starting from the seminal work of Volpano and Smith (VS 1997), type systems have become a prominent approach for a practical enforcement of information flow policies

Secure information flow analysis

Basic notions

- The starting point in secure information flow analysis is the classification of program variables into different security levels
  - The most basic distinction is to classify some variables as L, meaning low security, public information; and
  - other variables as H, meaning high security, private information
- The security goal is to prevent information in H variables from being leaked improperly. We need to prevent information in H variables from flowing to L variables
- More generally, we might want a lattice of security levels, and we would wish to ensure that information flows only upwards in the lattice.
- For example, if $L \leq H$, then we would allow flows from L to L, from H to H, and from L to H, but we would disallow flows from H to L.

Secure information flow analysis

Illegal flows

- Let us consider some examples from (DD 1977), assuming that `secret:H` and `leak:L`
- Clearly illegal is an explicit flow `leak=secret;`
- On the other hand, the following should be legal: `secret = leak;` as should `leak=76318;`
- Also dangerous is an implicit flow:

  ```
  if ((secret % 2)==0)
  leak = 0;
  else leak = 1;
  ```

  This copies the last bit of `secret` to `leak`
- Arrays can lead to subtle information leaks. If array *a* is initially all 0, then the program

  ```
  a[secret] = 1;
  for (int i = 0; i < a.length; i++) {
  if (a[i] == 1)
  leak = i;
  }
  ```

  leaks `secret`

Information flow type systems

- Structured programs

$$\frac{\vdash e : k \quad k \leq \tau(x)}{\vdash x := e : \tau(x)} \; \textit{Direct flows}$$

$$\frac{\vdash e : k \quad \vdash c_1 : k_1 \quad \vdash c_2 : k_2 \quad k \leq k_1, k_2}{\vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \; : k} \; \textit{Implicit flows}$$

- Unstructured programs

$$\frac{P(i) = \textbf{load}(x)}{i \vdash st \Rightarrow \tau(x) :: st} \qquad \frac{P(i) = \textbf{store}(x) \quad k \leq \tau(x)}{i \vdash k :: st \Rightarrow st}$$

$$\frac{P(i) = \textbf{ifeq}(j) \quad \forall j \in region(i), k \leq se(j)}{i \vdash k :: st \Rightarrow lift(k, st)}$$

Cache leakage

- Latency between cache hits and misses
- Attacks can be designed to recover cryptographic keys:
  - Tromer *et al* (TOS 2010), and Gullasch *et al* (GBK 2011) show efficient attacks on AES implementations
- In some cases the cryptographic key can be found without knowledge of either the cipher or plain text
- These attacks are based on the access of look-up tables: bits of the key can be deduced from the memory addresses accessed by the victim

Many adversary models: synchronous, access-driven, trace-based . . .

Example cache attack

Example cache attack

1. The attacker fills the cache with its own entries

Example cache attack

1. The attacker fills the cache with its own entries
2. It lets the victim run for a short time

Example cache attack

1. The attacker fills the cache with its own entries
2. It lets the victim run for a short time
3. The victim will access just a few table entries, which will replace some of the cache entries

Example cache attack

1. The attacker fills the cache with its own entries
2. It lets the victim run for a short time
3. The victim will access just a few table entries, which will replace some of the cache entries
4. The attacker measures the time to access *its own* addresses

Example cache attack

1. The attacker fills the cache with its own entries
2. It lets the victim run for a short time
3. The victim will access just a few table entries, which will replace some of the cache entries
4. The attacker measures the time to access *its own* addresses
5. After enough measures, a statistical analysis can be performed to recover the full key

Existing Countermeasures

- Some existing countermeasures:
  - Do not use the cache
  - Flush the cache
  - Dedicated cryptographic hardware
  - Application level countermeasures
    - Constant-time implementation
- Many of them have drawbacks:
  - Significant performance overhead
  - Specific to some classes of computations
  - Difficult to deploy, due to hardware requirements
- *"Finding an efficient solution that is application and architecture independent remains an open problem"*. Tromer, Osvik and Shamir (TOS 2010).

Constant time crypto algorithms

- Constant time algorithms:
  - do not branch on secrets
  - do not perform memory accesses that depend on secrets
- There are constant-time implementations of many cryptographic algorithms:
  - AES
  - DES
  - RSA
  - etc
- There was no rigorous proof that constant-time algorithms are protected to cache-based attacks when executed in virtualization platforms
- Many cryptographic implementations make array accesses that depend on secret keys, for efficiency

StealthMem

- StealthMem was presented by Erlingsson and Abadi in (EA 2007); and implemented by Kim, Peinado and Mainar-Ruiz (KPM 2012).
- Mechanism designed to protect a critical region of memory against cache side-channels in the cloud.
- Modify the hypervisor implementation to guarantee that stealth pages are never evicted from the cache.
- Benefits:
  - Minimal performance overhead
  - Compatibility with commodity hardware

StealthMem - Challenges

Does it work?

StealthMem does not provide *formal* guarantees of non-leakage of data allocated in stealth memory pages.

Correct usage

StealthMem requires manual modification of application code, to call the new StealthMem primitives.

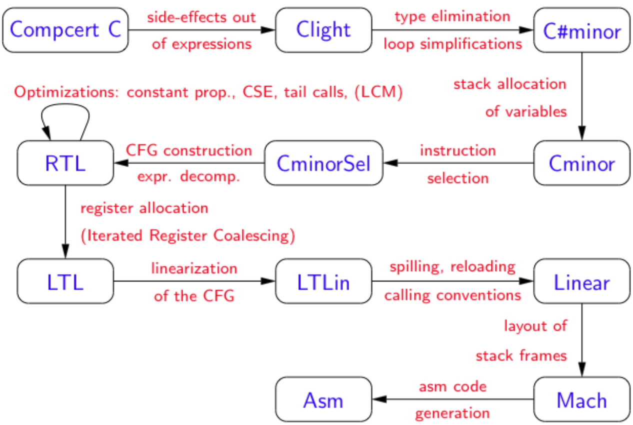
Static analysis of constant-time crypto algorithms

- Define a static analysis for enforcing constant-time on x86 programs
- Derive strong semantical guarantees for the class of programs accepted by our analysis (eg. no *cache-leakage*)
- Analyze realistic C programs, using the *CompCert* framework
- Do the analysis at a very low intermediate language, after all compiler optimizations.

CompCert

X. Leroy, INRIA - Rocquencourt, 2006

- C optimizer compiler developed in Coq
- Formal guarantees of semantic preservation
- Framework to formally reason about program semantics
- Will be used to perform the taint analysis on programs



MachIR Semantics

$$\frac{p[n] = \mathrm{op}(op, \vec{r}, r, n')}{(n, \rho, \mu) \xrightarrow{\emptyset} (n', \rho[r \mapsto [\![op]\!](\rho, \vec{r})], \mu)}$$

$$\frac{\begin{array}{c} p[n] = \mathrm{load}_{\varsigma}(addr, \vec{r}, r, n') \\ [\![addr]\!](\rho, \vec{r}) = v_{\mathrm{addr}} \qquad \mu[v_{\mathrm{addr}}]_{\varsigma} = v \end{array}}{(n, \rho, \mu) \xrightarrow{\mathtt{read}\ v_{\mathrm{addr}}} (n', \rho[r \mapsto v], \mu)}$$

$$\frac{\begin{array}{c} p[n] = \mathrm{store}_{\varsigma}(addr, \vec{r}, r, n') \\ [\![addr]\!](\rho, \vec{r}) = v_{\mathrm{addr}} \qquad store(\mu, \varsigma, v_{\mathrm{addr}}, \rho(r)) = \mu' \end{array}}{(n, \rho, \mu) \xrightarrow{\mathtt{write}\ v_{\mathrm{addr}}} (n', \rho, \mu')}$$

A Type system for constant-time

Generics

- Type-based information flow analysis that checks whether a MachIR program is constant-time, i.e. its control flow and its sequence of memory accesses do not depend on secrets
- To track how dependencies evolve during execution, the information flow analysis must be able to predict the set of memory accesses that each instruction will perform at runtime: Alias analysis
- Information flow type system

A Type system for constant-time

Alias (points-to) type system

$$
\begin{array}{llll}
\textit{alias} & ::= & & \\
 & \mid & \mathrm{Num} & \text{numerical value} \\
 & \mid & \mathrm{Symb}(\mathcal{S}) & \text{points to any cell allocated} \\
 & & & \text{for symbol } \mathcal{S} \\
 & \mid & \mathrm{Stack}(\delta) & \text{points to the } \delta^{\mathrm{th}} \text{ stack cell}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{A}[\![\mathrm{indexed}]\!](a, [r_1; r_2]) & = & \mathcal{A}[\![+]\!]([a(r_1); a(r_2)]) \\
\mathcal{A}[\![\mathrm{global}(S)]\!](a, \vec{r}) & = & \mathrm{Symb}(S) \\
\mathcal{A}[\![\mathrm{stack}(\delta)]\!](a, []) & = & \mathrm{Stack}(\delta) \\
 & = & \mathrm{Num} \ \ \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{A}[\![\mathrm{addrof}(\mathit{addr})]\!](a, \vec{r}) & = & \mathcal{A}[\![\mathit{addr}]\!](a, \vec{r}) \\
\mathcal{A}[\![\mathrm{move}]\!](a, [r]) & = & a(r) \\
\mathcal{A}[\![\mathrm{arith}(o)]\!](a, \vec{r}) & = & \mathcal{A}[\![o]\!](a[\vec{r}])
\end{array}
$$

$$
\frac{\mathcal{A}[\![\mathit{addr}]\!](A[n], \vec{r}) = \mathrm{Symb}(S) \qquad A[n][r \mapsto A[n](S)] \subseteq A[n']}{A \vdash n : \mathrm{load}_\varsigma(\mathit{addr}, \vec{r}, r, n')}
$$

$$
\frac{\mathcal{A}[\![\mathit{addr}]\!](A[n], \vec{r}) = \mathrm{Stack}(\delta)\} \qquad A[n][r \mapsto A[n](\delta)] \subseteq A[n']}{A \vdash n : \mathrm{load}_\varsigma(\mathit{addr}, \vec{r}, r, n')}
$$

$$
\frac{A[n] \subseteq A[n']}{A \vdash n : \mathrm{goto}(n')} \qquad \frac{A[n] \subseteq A[n_{then}] \quad A[n] \subseteq A[n_{else}]}{A \vdash n : \mathrm{cond}(c, \vec{r}, n_{then}, n_{else})}
$$

A Type system for constant-time

Information flow type system

$$\frac{p(n) = \mathrm{op}(op, \vec{r}, r, n')}{X_h \vdash n : \tau \Rightarrow \tau[r \mapsto \tau(\vec{r})]}$$

$$\frac{\begin{array}{c} p(n) = \mathrm{load}_\varsigma(addr, \vec{r}, r, n') \\ PointsTo(n, addr, \vec{r}) = \mathrm{Symb}(\mathcal{S}) \qquad \tau(\vec{r}) = \mathrm{Low} \end{array}}{X_h \vdash n : \tau \Rightarrow \tau[r \mapsto X_h(\mathcal{S})]}$$

$$\frac{p(n) = \mathrm{load}_\varsigma(addr, \vec{r}, r, n') \qquad PointsTo(n, addr, \vec{r}) = \mathrm{Stack}(\delta)}{X_h \vdash n : \tau \Rightarrow \tau[r \mapsto \tau(\delta) \sqcup \cdots \sqcup \tau(\delta + \varsigma - 1)]}$$

$$\frac{\begin{array}{c} p(n) = \mathrm{store}_\varsigma(addr, \vec{r}, r, n') \\ PointsTo(n, addr, \vec{r}) = \mathrm{Symb}(\mathcal{S}) \qquad \tau(\vec{r}) = \mathrm{Low} \qquad \tau(r) \sqsubseteq X_h(\mathcal{S}) \end{array}}{X_h \vdash n : \tau \Rightarrow \tau}$$

$$\frac{p(n) = \mathrm{store}_\varsigma(addr, \vec{r}, r, n') \qquad PointsTo(n, addr, \vec{r}) = \mathrm{Stack}(\delta)}{X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]}$$

$$\frac{p(n) = \mathrm{goto}(n')}{X_h \vdash n : \tau \Rightarrow \tau}$$

Definition of constant-time programs

A program p is constant-time with respect to a set of variables X_h^0, written $X_h^0 \vdash p$, if there exists (X_h, T) such that for every $\mathcal{S} \in X_h^0$, $X_h(\mathcal{S}) =$ High and for all nodes n and all its successors n', there exists τ such that

$$X_h \vdash n : T(n) \Rightarrow \tau \quad \wedge \quad \tau \sqsubseteq T(n')$$

where $\sqsubseteq$ is the natural lifting of $\sqsubseteq$ from $\mathbb{L}$ to types.

We automatically infer X_h and T using Kildall's algorithm

Information flow type system for S-constant time

$$\frac{p(n) = \mathrm{load}_{\varsigma}(addr, \vec{r}, r, n') \qquad PointsTo(n, addr, \vec{r}) = \mathrm{Symb}(\mathcal{S}) \qquad \tau(\vec{r}) = \mathrm{High} \implies \mathcal{S} \in X_s}{X_s, X_h \vdash n : \tau \Rightarrow \tau[r \mapsto \tau(\vec{r}) \sqcup X_h(\mathcal{S})]}$$

$$\frac{p(n) = \mathrm{store}_{\varsigma}(addr, \vec{r}, r, n') \qquad PointsTo(n, addr, \vec{r}) = \mathrm{Symb}(\mathcal{S}) \qquad \tau(\vec{r}) = \mathrm{High} \implies \mathcal{S} \in X_s \qquad \tau(\vec{r}) \sqcup \tau(r) \sqsubseteq X_h(\mathcal{S})}{X_s, X_h \vdash n : \tau \Rightarrow \tau}$$

Soundness of Constant-Time Type System

- Establishes a non-interference property based on the semantics of MachIR programs
- Based on an equivalence relation between states ($s \sim_{X_h,\tau} s'$).
- Extend the equivalence to execution traces ($\theta \sim_{X_h,T} \theta'$)
- We can prove that all programs that type-check have the same control flow and memory accesses:

$$X_h^0 \vdash p \quad \wedge \quad s \sim_{X_h,T(pc_0)} s' \quad \Longrightarrow \quad \theta \sim_{X_h,T} \theta'$$

Automatic vulnerability analysis of crypto-algorithms

We successfully evaluate our approach based on a representative set of off-the-shelf implementations of cryptographic algorithms, including:

- the PolarSSL implementations of AES, DES, Blowfish and RC4, and the ECRYPT implementation of SNOW, which are vulnerable to cache-based attacks on standard platforms;
- oblivious cryptographic algorithms, including SHA256, TEA and Salsa20.

EXAMPLE	LOC	# ADDRESSES	SIZE (KB)
DES	836	10	2
Blowfish	279	1	4
AES	744	5	4
RC4	164	1	0.25
Snow	757	6	6
Salsa20	1077	0	0
TEA	70	0	0
SHA256	419	0	0

Conclusions

- Constant-time cryptography is an oft advocated solution against cache-based attacks. We have:
  - developed an automated analyzer for constant-time cryptography
  - given the first formal proof that constant-time programs are indeed protected against concurrent cache-based attacks.
- We have extended our analysis to the setting of stealth memory:
  - we have developed the first formal security analysis of stealth memory.
  - our results have been formalized in the Coq proof assistant.
- Our analyses have been validated experimentally on a representative set of algorithms.
- The paper System-level non-interference for constant-time cryptography was accepted in ACM CCS 2014

References

Gary A. Kildall.
A unified approach to global program optimization.
Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '73, pp. 194-206, ACM,1973.

D. Denning, P. Denning.
Certification of programs for secure information flow.
Communications of the ACM, 20(7):504-513, 1977.

D. Volpano, G. Smith.
A Type-Based Approach to Program Security.
Proceedings of TAPSOFT '97, pp 607-621, 1997.

E. Tromer, D. A. Osvik, A. Shamir.
Efficient Cache Attacks on AES, and Countermeasures.
Journal of Cryptology, volume 23, pp. 37-71, 2010.

D. Gullasch, E. Bangerter, S. Krenn.
Cache Games - Bringing Access-Based Cache Attacks on AES to Practice.
Proceedings of S&P 2011: IEEE Symposium on Security and Privacy, pp. 490-505, 2011.

U. Erlingsson, M. Abadi.
Operating system protection against side-channel attacks that exploit memory latency.
Technical Report MSR-TR-2007-117, Microsoft Research, 2007.

T. Kim, M. Peinado, G. Mainar-Ruiz.
STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud.
Proceedings of USENIX Security 2012, pp. 11-21, 2012.

Referencias (Cont.)

G. Barthe, G. Betarte, J.D. Campo, C. Luna.
Formally verifying isolation and availability in an idealized model of virtualization.
Proceedings of FM2011: 17th International Symposium on Formal Methods, Lecture Notes in Computer Science, vol. 6664, pp 231-245, Ireland, June 2011.

G. Barthe, G. Betarte, J.D. Campo, C. Luna.
Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization.
Proceedings of CSF 2012: 25th IEEE Computer Security Foundations Symposium, pp. 186-197, IEEE Computer Society Press, 2012.

G. Barthe, G. Betarte, J.D. Campo, J. Chimento, C. Luna.
Formally verified implementation of an idealized model of virtualization.
En Post-Proceedings of TYPES 2013: Workshop on Types for Proofs and Programs. To appear in LIPcs, 2014.

G. Barthe, G. Betarte, J.D. Campo, C. Luna, D. Pichardie.
System-level non-interference for constant-time cryptography.
To appear in Proceedings of CCS 2014: the 21st ACM Conference on Computer and Communications Security, November 2014.

Página del proyecto VirtualCert
`www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`