On Protection by Layout Randomization

Martín Abadi Gordon Plotkin

COLLOQUE ANNIVERSAIRE
EN L'HONNEUR DE
GÉRARD BERRY ET JEAN-JACQUES LÉVY

Outline

Topic

- Our main theme concerns formal proofs of security, both in computational models and in symbolic models.
- This talk is about low-level software protection, rather than cryptographic protocols.
- More could be done in this direction, perhaps learning from work on protocols.

Low-Level Attacks and Protection

- Many attack techniques:
  - Buffer overflows
  - Exception overwrites
  - return-to-libc
  - jump-to-libc
  - use-after-free attacks
- Often with knowledge and control of the heap

- Many defenses:
  - Stack canaries
  - Safe exception handling
  - NX (No eXecute) data
  - Layout randomization
- Useful mitigations
- But not necessarily perfect in a precise sense
- Nor all well understood

Executing Existing Code

- With NX defenses, attackers cannot simply inject data and then run it as code.
- But attackers can still run existing code:
  - an existing function, such as system(),
  - even code in the middle of a function,
  - even “accidental” code (e.g., starting half-way in a long x86 instruction),
  - even dead code.

Layout Randomization

- Attacks often depend on addresses.
  $\Longrightarrow$ *Let us randomize the addresses!*
  - Considered for data at least since the rise of large virtual address spaces (e.g., [Druschel & Peterson,1992] on fbufs).
  - Now present in Linux (PaX) and Windows (ASLR).

Goals

- Define a form of layout randomization.
- Study it precisely.
- Draw on ideas from the theory of programming languages and security, where:
  - there is much previous work on using equivalences for security properties (e.g., in information-flow control and in the study of protocols).
  - and there is work on preservation of security properties, perhaps given by such equivalences, by compilation.
- *Not* (yet) model and prove the security of actual systems.

Private and Public Variables

- We distinguish private variables (which should not be directly accessible to attackers) from public ones.
- Privacy gives secrecy and integrity properties:
  - If l is private, then $l := c$ should not reveal c.
  - If l is private, then an attacker should not be able to tamper with l in

$$\begin{array}{l}\lambda f\!:\!\mathtt{nat} \to \mathtt{unit}.\\ \quad l := c;\\ \quad f(c);\\ \quad \mathtt{if}\; !l = c\; \mathtt{then}\; l' := c\; \mathtt{else}\; l' := c'\end{array}$$

Implementation

- l is mapped to a natural-number address $w(l)$, where w is a *layout* function.
- So, for example, $l := c$ compiles to $w(l) := c$

High- and Low-Level Threats

A high-level attacker may

- access public variables,
- call functions, pass arguments.

A low-level attacker may in addition access memory through natural-number addresses. There are two models:

- Fatal Error Model A fatal error is raised on an erroneous low-level memory address (as may happen in kernel mode).
- Recoverable Error Model A recoverable error is raised on an erroneous low-level memory address (as may happen in user mode).

Implementation (cont.)

- l is mapped to a natural-number address $w(l)$, where w is a *layout* function (w is fixed on public locations).
- So, for example, $l := c$ compiles to $w(l) := c$
- If an attacker can guess w, then it can break the expected security properties.
- Choosing w randomly may preserve those properties.
- We may expect strong guarantees for the fatal error model, but we may only expect guarantees parametrized on some count of memory accesses for the recoverable error model.

Approach

- Define high-level and low-level languages.
  $\longrightarrow$ View the implementations as translations.
- View attackers as contexts.
  $\longrightarrow$ Study whether low-level contexts correspond to high-level contexts.
- Phrase security properties as equivalences.
  $\longrightarrow$ Study whether equivalences are preserved.
- In this talk we only consider the recoverable error model.

The Source Language

- Higher-order lambda calculus,
- with reading and writing operations on locations that hold natural numbers,
- and with standard base types and a type of locations.

Syntax: Types

- These are:

$$\sigma ::= \texttt{loc} \mid \texttt{nat} \mid \texttt{unit} \mid \sigma \times \tau \mid \sigma + \tau \mid \sigma \to \tau$$

- Commands are terms of type `unit`; they are evaluated only for their side-effects.
- We don't need a boolean type as we can take `bool` to be $\texttt{unit} + \texttt{unit}$.

Syntax: Terms

- Terms are ranged over by *M*, *N*, *P*, ...

M	$::=$	$x \mid$	Variables
		$l \; (l \in \mathrm{Loc}) \mid\; ! \mid := \mid c \mid$	Constants
		$(M, N) \mid \mathtt{fst}\, M \mid \mathtt{snd}\, M \mid$	Products
		$\mathtt{inl}\, M \mid \mathtt{inr}\, M \mid$	Sums
		$\mathtt{cases}\, P\, \mathtt{inl}\, x{:}\sigma.\, M\, \mathtt{inr}\, y{:}\tau.\, N \mid$	
		$\lambda x{:}\sigma.\, M \mid MN \mid$	Functions
		$\mathtt{letrec}\, f(x{:}\sigma) = M\, \mathtt{in}\, N$	Recursion

- Loc is a finite set of locations; it is divided into sets PubLoc and PriLoc of public and private locations .
- *c* ranges over arithmetic operations and predicates.
- Programs are simply terms with no free variables.

Syntax: Typing

- Terms are given types $M : \sigma$ using typing axioms and rules.
- Some example typing axioms for constants:

$$l : \mathtt{loc}\ (l \in \mathit{Loc}) \qquad ! : \mathtt{loc} \to \mathtt{nat} \qquad := : \mathtt{loc} \times \mathtt{nat} \to \mathtt{unit}$$

- Some example typing rules

$$\frac{M : \sigma \quad N : \tau}{(M, N) : \sigma \times \tau} \qquad \frac{M : \sigma \times \tau}{\mathtt{fst}\ M : \sigma}$$

Contexts and Attackers

- A context is a program of type

$$\sigma \to \texttt{bool}$$

- A public context is one that cannot access private locations directly. That is, it contains no locations in PriLoc.
- For us:

$$\text{attackers} = \text{public contexts}$$

Full Abstraction

- Given a language L, two programs M and N are observationally equivalent

$$M \approx_L N$$

if they have the same observable behavior in all contexts.

- A translation $M \mapsto M^{\downarrow}$ from L to another language L' is fully abstract if it preserves and reflects observational equivalence:

$$M \approx_L N \quad \text{iff} \quad M^{\downarrow} \approx_{L'} N^{\downarrow}$$

Full abstraction (preview)

- We adapt these ideas to public contexts.
- We also weaken the low-level equivalences, with probabilities and complexity bounds.
- Modulo these changes, we do obtain full abstraction
  - The results are specific to certain languages.
  - They fail for some subsets and supersets.

Source Language: Semantics

- We define a transition relation

$$(s, M) \to (s', M')$$

  by axioms and rules.
- Example axioms:

$$(s, !l) \longrightarrow (s, n) \quad (s(l) = n)$$
$$(s, l := n) \longrightarrow (s[l \mapsto n], \texttt{skip})$$

- Computations are then either finite:

$$(s_1, M_1) \to (s_2, M_2) \to \ldots \to (s, V) \quad \equiv_{\text{def}} \quad (s_1, M_1) \Rightarrow (s, V)$$

  where V is a value, e.g., l, 3, $(4, \texttt{inl}\, 3)$, or $\lambda x\!:\!\texttt{nat}.\, x + x$.
- or infinite:

$$(s_1, M_1) \to (s_2, M_2) \to \ldots \quad \equiv_{\text{def}} \quad (s_1, M_1) \Uparrow$$

Source Language: Evaluation Function

- Evaluation functions give the observable behaviour of programs.
- Eval is the high-level evaluation function; it is defined by:

$$\mathrm{Eval}(M, s) = \begin{cases} (s', V) & \text{if } (s, M) \Rightarrow (s', V) \\ \Omega & \text{if } (s, M) \Uparrow \end{cases}$$

- So if termination is normal, Eval maps a program and an initial store to the final store and resulting value.

Source Language: Observational Equivalence

Two high-level programs are publicly observationally equivalent if no public context can distinguish them:

For M, N of type σ,

$$M \approx_{h,p} N$$

iff for every public context $C : \sigma \to \texttt{bool}$ and store s,

(1) either both $\mathrm{Eval}(CM, s)$ and $\mathrm{Eval}(CN, s)$ diverge,
(2) or else they both yield the same result value and two new publicly-equivalent stores (i.e., two stores that coincide on PubLoc).

Equivalences (cont.)

Secrecy and integrity properties can be phrased as public equivalences:

$$l := c \quad \approx_{h,p} \quad l := c'$$

and

$$\begin{array}{l} \lambda f : \mathtt{nat} \to \mathtt{unit}. \\ \quad l := c; \\ \quad f(c); \\ \quad \mathtt{if}\ !l = c\ \mathtt{then}\ l' := c\ \mathtt{else}\ l' := c' \end{array} \quad \approx_{h,p} \quad \begin{array}{l} \lambda f : \mathtt{nat} \to \mathtt{unit}. \\ \quad l := c; \\ \quad f(c); \\ \quad l' := c \end{array}$$

The Target Model, Informally

- A layout w is a function of type $\mathrm{Loc} \hookrightarrow \{0, \ldots, c\}$ chosen uniformly at random.
- A memory m is a function of type $\{0, \ldots, c\} \longrightarrow \mathbb{N} + 1$
  - Memory may be accessed directly through natural-number addresses.
  - Some addresses may be unused.
- Accesses to unused addresses are recoverable errors

Target Language

- The types are:

$$\sigma ::= \cancel{\texttt{loc}} \mid \texttt{nat} \mid \texttt{unit} \mid \sigma \times \tau \mid \sigma + \tau \mid \sigma \to \tau$$

- The terms are as before, but the memory access constants are now typed differently:

$$l : \texttt{nat} \quad (l \in \mathbf{Loc})$$
$$! : \texttt{nat} \to \texttt{nat} + \texttt{unit}$$
$$:= : \texttt{nat} \times \texttt{nat} \to \texttt{com} + \texttt{unit}$$

- The "$+\,\texttt{unit}$" represents the recoverable error case. We write $\texttt{error}$ for $\texttt{inr}\,*$.

Exploring the Memory

Sample programs that explore memory:

$$\texttt{cases}\,!0\,\texttt{inl}\,x\texttt{:nat.}\,x\quad \texttt{inr}\,y\texttt{:unit.}\,!1$$

$$\texttt{cases}\,!0\,\texttt{inl}\,x\texttt{:nat.}\,x\quad \texttt{inr}\,y\texttt{:unit.cases}\,!1\,\texttt{inl}\,x\texttt{:nat.}\,x\quad \texttt{inr}\,y\texttt{:unit.}\,!2$$

$\longrightarrow$ Results for the recoverable error model require bounds on the number of accesses.

Our semantics therefore takes such accesses into account.

Semantics

- The semantics is given by axioms and rules for two transition relations, each relative to a layout:

$$w \models (m, M) \to (m', M') \qquad w \models (m, M) \xrightarrow{a} (m', M')$$

- with the latter indicating a faulty memory address at a.
- Some semantics:

$$w \models (m, l) \longrightarrow (m, m(w(l))) \quad (l \in \mathrm{Loc})$$
$$w \models (m, !a) \longrightarrow (m, \texttt{inl}\ n) \qquad (\text{if } a \in \{0, \ldots, c\} \text{ and } m(a) = n)$$
$$w \models (m, !a) \xrightarrow{a} (m, \texttt{error}) \qquad (\text{if } a \notin \{0, \ldots, c\} \text{ or } m(a) = *)$$
$$w \models (m, a := n) \longrightarrow (m[a \mapsto n], \texttt{inl skip}) \qquad (\text{if } a \in \{0, \ldots, c\} \text{ and } m(a) \neq *)$$
$$w \models (m, a := n) \xrightarrow{a} (m, \texttt{error}) \qquad (\text{if } a \notin \{0, \ldots, c\} \text{ or } m(a) = *)$$

(recalling that we write `error` for $\texttt{inr}\,*\texttt{:nat} + \texttt{unit}$)

Computations

- As before, computations are then either finite:

$$(m_1, M_1) \xrightarrow{(a_1)} \ldots \xrightarrow{(a_{n-1})} (m_n, V) \quad \equiv_{\text{def}} \quad (m_1, M_1) \stackrel{A}{\Longrightarrow} (m, V)$$

  where (a) indicates a possible a and A is the set of the actually occurring such a's.

- or infinite:

$$(m_1, M_1) \xrightarrow{(a_1)} \ldots \xrightarrow{(a_{n-1})} (m_n, V) \xrightarrow{(a_n)} \quad \equiv_{\text{def}} \quad (m_1, M_1) \Uparrow^A$$

Evaluation Function & Faulty Memory Accesses

- Eval_w^b is the low-level evaluation function; it has a layout parameter w, and a memory-access bound, b:

$$\mathrm{Eval}_w^b(M, m) = \begin{cases} (m', V) & \text{if } w \models (m, M) \stackrel{A}{\Longrightarrow} (m', V) \\ & \text{and } |A \cap \{0, \ldots, c\}| \leq b, \text{ for some } A \\ \Omega & \text{otherwise} \end{cases}$$

- When it terminates normally, and within bounds, it maps a program and initial memory to a final memory and value.
- It is also useful to record the faulty memory accesses within bounds, setting:

$$\mathrm{Acc}_w(M, m) = A \cap \{0, \ldots, c\}$$

where either $(m, M) \stackrel{A}{\Longrightarrow} -$ or $(m, M) \Uparrow^A$.

Translations

- With each high-level program $M : \sigma$ we associate a low-level program $M^{\downarrow} : \sigma^{\downarrow}$.
  Essentially this just replaces `loc` by `nat`.
- With each low-level context $C : \sigma \to \texttt{bool}$ we associate a high-level context $C^{\uparrow} : \sigma \to \texttt{bool}$.
  Essentially this just replaces non-public memory accesses by recoverable errors.

Probability Bound

- δ_b is the probability that b distinct probes do not hit any private location of a layout chosen uniformly at random.
- We have:

$$\delta_b = \frac{|w : \mathrm{PriLoc} \hookrightarrow \{1, \ldots, a-b\}|}{|w : \mathrm{PriLoc} \hookrightarrow \{1, \ldots, a\}|}$$

  where $a =_{\mathrm{def}} (c+1) - |\mathrm{PubLoc}|$ is the amount of memory available for private locations (and assuming $b \leq a$).
- In case $|\mathrm{PriLoc}| = 1$ this simplifies to:

$$\delta_b = 1 - b/a$$

Results for the Recoverable-Error Model: Theorem on Attackers

- Suppose $C : \sigma \to \texttt{bool}$ is a low-level attacker , i.e., a public low-level term (so σ is $\texttt{loc}$-free). We show that $C^{\uparrow} : \sigma \to \texttt{bool}$ is a high-level attacker of equal power, probabilistically.
- The theorem bounds the probability that the semantics of $C^{\uparrow}M$ and $CM^{\downarrow}$ coincide, for a randomly chosen layout w.
- There is an assumption on the number b of erroneous accesses: without this bound, an attacker could explore all of memory.
- For small b, the high- and low-level semantics coincide the most, with probability close to 1 when c is sufficiently large.

Statement of Theorem

Theorem

Let $M : \sigma$ be a high-level term and let $C : \sigma \to \texttt{bool}$ be a low-level attacker. Then, for any store s, and $0 \leq b \leq c - |\mathrm{Loc}|$, one of the following holds:

1. $\mathrm{Prob}(|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| > b) \geq \delta_{b+1}$, *or*
2. $\mathrm{Prob}(|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| \leq b$ *and* $\mathrm{Eval}(C^{\uparrow}M, s)_w = \mathrm{Eval}^b_w(CM^{\downarrow}, s_w)) \geq \delta_b$.

These alternatives are mutually exclusive if $\delta_{b+1} > 1/2$.

Remark: We would like to improve δ_{b+1} to δ_b here, and below, by considering *possibly* faulty memory accesses.

Low Level Partial Observational Equivalence

We set

$$M \approx^{b}_{l,p} N$$

iff for all low-level public contexts $C : \sigma \to \texttt{bool}$ and stores s:

- Either, for some $s' \restriction \mathrm{PubLoc} = s'' \restriction \mathrm{PubLoc}$ and V:

$$\mathrm{Prob}(\mathrm{Eval}^{b}_{w}(CM, s_w) = (s'_w, V)) \geq \delta_b$$

and

$$\mathrm{Prob}(\mathrm{Eval}^{b}_{w}(CN, s_w) = (s''_w, V)) \geq \delta_b$$

- or:

$$\mathrm{Prob}(\mathrm{Eval}^{b}_{w}(CM, s_w) = \Omega) \geq \delta_{b+1}$$

and

$$\mathrm{Prob}(\mathrm{Eval}^{b}_{w}(CN, s_w) = \Omega) \geq \delta_{b+1}$$

Theorem

Full Abstraction for Recoverable Error Model

Let $M, N : \sigma$ *be high-level terms. Then, assuming that* σ *is* `loc`*-free,* $0 \leq b \leq c - |\mathrm{Loc}|$*, and* $\delta_{b+1} > 1/2$*, we have:*

$$M \approx_{h,p} N \quad \textit{iff} \quad M^{\downarrow} \approx^{b}_{l,p} N^{\downarrow}$$

Further Work: Passing Locations

- Our results apply to passing functions that manipulate locations,
  e.g., $(\lambda x{:}\mathtt{unit}.\,!l, \lambda x{:}\mathtt{nat}.\,l := x)$
- but not at all to passing locations,
  e.g., two naked private locations l, l'

An Example (Poisoning)

An implementation could be “poisoned” with a natural number that does not represent a location, e.g.,:

$$
\begin{array}{lcl}
M & = & \lambda x{:}\mathtt{loc}.\,3 \\
N & = & \lambda x{:}\mathtt{loc}.\,\mathtt{let}\ y\ \mathtt{be}\ !x\ \mathtt{in} \\
 & & \qquad \mathtt{cases}\ y\ \mathtt{inl}\ x{:}\mathtt{nat}.\,3\ \mathtt{inr}\ y{:}\mathtt{unit}.\,5
\end{array}
$$

are high- but not low-level publicly observationally equivalent.

An Example (Stateful Secrecy)

Here, in an implementation, an attacker could pass an f that stores its input, a secret location, for future use:

$$\begin{aligned} M &= \lambda f{:}\mathtt{loc} \to \mathtt{unit}. \\ &\quad \mathtt{if}\,!l_2 = 0\,\mathtt{then}\,l_2 := 1; f(l_1)\,\mathtt{else}\,\Omega \\ &\quad l_1 := 0 \\ M' &= \lambda f{:}\mathtt{loc} \to \mathtt{unit}. \\ &\quad \mathtt{if}\,!l_2 = 0\,\mathtt{then}\,l_2 := 1; f(l_1)\,\mathtt{else}\,\Omega \\ &\quad l_1 := 1 \end{aligned}$$

However at high level this is not possible, and as f can be called at most once, M and M' are publicly observationally equivalent.

Further Work (cont.)

- Our languages lack some features:
  - code pointers,
  - dynamic allocation,
  - ...
- Further work could discuss more attacks:
  - jump-to-libc attacks,
  - use-after-free attacks,
  - ...
- In another direction, further work could examine more features of implementations, e.g., replication.
- It would also be good to have methods to prove public equivalences $M \approx_{h,p} N$

Félicitations à Gérard et Jean-Jacques !