言語の security と full abstraction について

Sumii

1998 年 6 月 11 日

1 はじめに

types メーリングリスト[1]に流れてきた Abadi の論文 [1] を紹介しないか、というお達しがあったので、関連研究と一緒にサーベイしてみました。大きく分けて Java 関係の話 [3] と secure calculi 関係の話 [2] の二つがあったのですが、前者まで調べる時間的余裕がなかったので、今回は後者について扱いたいと思います。

2 secure な通信の実現

π-calculus において、新しいチャネル n を作って、値 m を受け渡すプロセス

$$P = (\nu n)(\overline{n}\langle m\rangle \mid n(x)\ .\ \mathbf{0})$$

を考えてみます。 restriction (νn) のおかげで、 n はいわば private なチャネルになっています。すなわち、外部から見るとこのプロセスは

$$P' = (\nu n)(\overline{n}\langle m'\rangle \mid n(x)\ .\ \mathbf{0})$$

というプロセス (m' は任意の値) や、何もしないプロセス $\mathbf{0}$ と理論的には区別できないはずです。

ところが、現実に n のような private チャネルを実装しようとしても、 $\overline{n}\langle m\rangle$ と $n(x)\ .\ \mathbf{0}$ が (インターネットのような) 必ずしも信頼できないネットワーク上の異なるマシン上に存在する場合、素朴な方法だと P と P' や $\mathbf{0}$ が区別できてしまうおそれがあります。たとえば、 秘密鍵 n を用いて m を暗号化し、 public なチャネル c 上で受け渡す、という実装 (spi-calculus で書くと

$$[\![P]\!] = (\nu n)(\overline{c}\langle\{m\}_n\rangle \mid c(y)\ .\ \mathbf{case}\ y\ \mathbf{of}\ \{x\}_n\ \mathbf{in}\ \mathbf{0})$$

となる) には、いくつもの「穴」があります。

Leak of traffic patterns

元のプロセス P は外部から見ると $\mathbf{0}$ と区別できないはずなのに、先の $[\![P]\!]$ は public なチャネル c 上で通信を行うので、 $c(z)\ .\ \cdots$ という形のプロセスによって、はっきりと区別できてしまいます。

これを防ぐためには、 c に常に「雑音」を流し続けるプロセス $!(\nu p)\overline{c}\langle\{\}_p\rangle$ を導入するという方法があります。

Trivial denial-of-service vulnerability

π-calculus において、

$$Q = (\nu n)(\overline{n}\langle m\rangle \mid n(x)\ .\ \overline{x}\langle\rangle)$$

[1] types@cs.indiana.edu。詳しくは http://www.cs.indiana.edu/hyplan/pierce/types/ を参照。

というプロセスは、$\overline{m}\langle\rangle$ と等価なはずです。しかし、これを (spi-calculus で) 素朴に

$$[\![Q]\!] = (\nu n)(\overline{c}\langle\{m\}_n\rangle \mid c(y)\ .\ \mathbf{case}\ y\ \mathbf{of}\ \{x\}_n\ \mathbf{in}\ \overline{c}\langle\{\}_x\rangle)$$

と実装すると、c にでたらめな値 p を送るプロセス $\overline{c}\langle p\rangle$ があったら、

$$\overline{c}\langle p\rangle \mid [\![Q]\!] \quad \rightarrow \quad (\nu n)(\overline{c}\langle\{m\}_n\rangle \mid \mathbf{case}\ p\ \mathbf{of}\ \{x\}_n\ \mathbf{in}\ \overline{c}\langle\{\}_x\rangle)$$

のようにデッドロックしてしまう可能性があります。

このような問題を防ぐためには、受信側を replicate する (! をつける) 方法が考えられます。[2]

Exposure to replay attacks

π-calculus では

$$R = (\nu n)(\overline{n}\langle m_1\rangle \mid \overline{n}\langle m_2\rangle \mid n(x)\ .\ \overline{x}\langle\rangle \mid n(x)\ .\ \overline{x}\langle\rangle)$$

と $\overline{m_1}\langle\rangle \mid \overline{m_2}\langle\rangle$ は外部から区別できないはずです。しかし、これを

$$[\![R]\!] = (\nu n)(\overline{c}\langle\{m_1\}_n\rangle \mid \overline{c}\langle\{m_2\}_n\rangle \mid c(y)\ .\ \mathbf{case}\ y\ \mathbf{of}\ \{x\}_n\ \mathbf{in}\ \overline{c}\langle\{\}_x\rangle \mid c(y)\ .\ \mathbf{case}\ y\ \mathbf{of}\ \{x\}_n\ \mathbf{in}\ \overline{c}\langle\{\}_x\rangle)$$

と実装してしまうと、c から奪った値を複製して流すプロセス $c(y)\ .\ (\overline{c}\langle y\rangle \mid \overline{c}\langle y\rangle)$ があったら、m_1 か m_2 に二回送信が行われてしまう可能性があります。

これに対する対処としては、メッセージに time stamp や sequence number をつける方法や、challenge-response 型のプロトコルを用いる方法などが考えられます。

Leak of message equalities

π-calculus の

$$S = (\nu n)(\overline{n}\langle m_1\rangle \mid \overline{n}\langle m_2\rangle \mid n(x)\ .\ \mathbf{0} \mid n(x)\ .\ \mathbf{0})$$

というプロセスを、

$$[\![S]\!] = (\nu n)(\overline{c}\langle\{m_1\}_n\rangle \mid \overline{c}\langle\{m_2\}_n\rangle \mid c(y)\ .\ \mathbf{case}\ y\ \mathbf{of}\ \{x\}_n\ \mathbf{in}\ \mathbf{0} \mid c(y)\ .\ \mathbf{case}\ y\ \mathbf{of}\ \{x\}_n\ \mathbf{in}\ \mathbf{0})$$

と実装すると、(前述の問題以外に) public なチャネル c に流れている暗号文 $\{m_1\}_n$ と $\{m_2\}_n$ を比較することによって、$m_1 = m_2$ かどうかを知ることができてしまうという問題があります。

対処法としては、送信側を $(\nu p_1)\overline{c}\langle\{m_1, p_1\}_n\rangle$ や $(\nu p_2)\overline{c}\langle\{m_2, p_2\}_n\rangle$ のようにして、適当な乱数をメッセージに混ぜる方法があります。[3]

Lack of forward secrecy

private なチャネル n を作り、一度使ってから公開する π-calculus のプロセス

$$T = (\nu n)(\overline{n}\langle m\rangle \mid n(x)\ .\ \overline{p}\langle n\rangle)$$

は、外から見れば単なる $(\nu n)\overline{p}\langle n\rangle$ と等価なはずです。しかし、これを素朴に

$$[\![T]\!] = (\nu n)(\overline{c}\langle\{m\}_n\rangle \mid c(y)\ .\ \mathbf{case}\ y\ \mathbf{of}\ \{x\}_n\ \mathbf{in}\ \overline{c}\langle\{n\}_p\rangle)$$

と実装すると、

$$c(x)\ .\ (\overline{c}\langle x\rangle \mid c(y)\ .\ \mathbf{case}\ y\ \mathbf{of}\ \{z\}_p\ \mathbf{in}\ \mathbf{case}\ x\ \mathbf{of}\ \{w\}_z\ \mathbf{in}\ \overline{d}\langle w\rangle)$$

[2] しかしながら、同様の問題は送信側についても起こるわけで、それも防ごうとするとちょっと複雑になります。

[3] 最初からそういう性質を持っている確率的な暗号系を使えば、そもそもこの問題は起こりません。

のように、「最初に流れた暗号文 x を記憶しておいて、後から公開された鍵 n を使って解読し、 public なチャネル d に流す」といったことができてしまいます。

これを防ぐのは容易ではありません。通信のたびに鍵をかえる方法も考えられますが、複雑になる上に非効率的です。ただし、 π-calculus のかわりに join-calculus を対象にすれば、そもそもチャネルから値を受け取る能力が委譲できないので、このような問題は起こりません。

3 実装の正しさと full abstraction

以上にあげたような問題は、基本的に実装の改良により回避できました。しかし、個々の問題に対して ad hoc な対処を行っても、それで十分かどうかは保証しきれません。 π-calculus (あるいは join-calculus) から spi-calculus (あるいは sjoin-calculus) への変換 $[\![\cdot]\!]$ が本当に「正しい」ということは、以下の条件によってあらわされます。

$$P \approx_\pi Q \quad \Leftrightarrow \quad [\![P]\!] \approx_{spi} [\![Q]\!] \quad \text{for any processes } P, Q$$

つまり、 $[\![\cdot]\!]$ によってプロセスの「等しさ」が損なわれないということです。これを full abstraction といいます。

spi-calculus や sjoin-calculus のような low-level な言語 (public なチャネル上の暗号化された通信を明示的に扱っている) の上で、 π-calculus や join-calculus のような high-level な言語 (private なチャネルを抽象的に扱っている) を実装する場合、 full abstraction は high-level な言語の上で reasoning を行う際、 low-level な言語における実装について気にする必要がない、ということを意味します。もちろん、 high-level な言語の上でまずいもの (秘密を漏らしてしまうプロセスなど) は、 low-level な言語に変換してもやはりまずいので、 full abstraction 自体は必ずしも security を保証しているわけではありません。

4 join-calculus から sjoin-calculus へ

具体的な成功例として、 join-calculus の sjoin-calculus における fully-abstract な実装 [2] について述べます。

The join-calculus and the sjoin-calculus

join-calculus と sjoin-calculus の syntax は Figure 1, 2 の通りです。 join-calculus の operational semantics は以前遠藤さんが説明してくださったのと同様です。 sjoin-calculus のほうは基本的には join-calculus と一緒なのですが、 **keys** x^+, x^- で暗号化鍵 x^+ と復号化鍵 x^- のペアを作り、 **decrypt** v **using** w **to** $\tilde{x}$ **in** P **else** P' で暗号文の復号を試みる (失敗したら P' を実行する) という操作が加わっており、それに対応する reduction rule として

$$\mathbf{keys}\ x^+, x^- \vdash \mathbf{decrypt}\ \{\tilde{v}\}_{x^+}\ \mathbf{using}\ x^-\ \mathbf{to}\ \tilde{y}\ \mathbf{in}\ P\ \mathbf{else}\ P' \quad \rightarrow \quad \mathbf{keys}\ x^+, x^- \vdash P\{\tilde{v}/\tilde{y}\}$$

と

$$\mathbf{keys}\ x^+, x^- \vdash \mathbf{decrypt}\ w\ \mathbf{using}\ x^-\ \mathbf{to}\ \tilde{y}\ \mathbf{in}\ P\ \mathbf{else}\ P' \quad \rightarrow \quad \mathbf{keys}\ x^+, x^- \vdash P' \qquad (w \neq \{\tilde{v}\}_{x^+})$$

があります。暗号化鍵と復号化鍵が等しいとき (秘密鍵暗号系のとき) は、 **keys** x^+, x^- を **key** x と略します。

The low-level network model

具体的な変換方法について述べる前に、前提となる untrusted なネットワークを以下のように定義します。

$$\begin{aligned} Env[\cdot] \quad \stackrel{\text{def}}{=} \quad & \mathbf{def}\ recv\langle\kappa\rangle \mid emit\langle m\rangle \rhd \kappa\langle m\rangle \\ & \mathbf{in}\ plug\langle emit, recv\rangle \mid W \mid \cdot \end{aligned}$$

$$W \stackrel{\text{def}}{=} \textbf{repl}\ (\textbf{def fresh}\ m\ \textbf{in repl}\ \mathit{emit}\langle m\rangle)$$

このネットワークの「使い方」ですが、メッセージを流したい人はそれを $emit$ に送り、メッセージを受けとりたい人はそのためのチャネルを作って $recv$ に送る、ということをします。 $plug$ という free name を通じて $emit$ と $recv$ を公開することにより、ネットワークが untrusted であることを表しています。

まっとうな受信者は、次のような「復号化に失敗した (= 自分に関係のない) メッセージはネットワークに戻す」という動作を行います。

$$\begin{aligned}\textbf{let}\ \tilde{y} = \textbf{filter}\ k\ \textbf{in}\ P \quad \stackrel{\text{def}}{=} \quad & \textbf{def}\ \kappa\langle m\rangle \quad \triangleright \quad \textbf{decrypt}\ m\ \textbf{using}\ k\ \textbf{to}\ \tilde{y}\ \textbf{in}\ P \\ & \qquad\qquad\qquad\qquad \textbf{else}\ \mathit{emit}\langle m\rangle \mid \mathit{recv}\langle\kappa\rangle \\ & \textbf{in}\ \mathit{recv}\langle\kappa\rangle\end{aligned}$$

Session-based protocol

このようにして定義した insecure なネットワーク上で、 secure な通信を行うためのプロトコルとして、論文では二つの例があげられているのですが、ここではその一方の challenge-response 型プロトコルについて述べます。

暗号化鍵 x^+ を使って値 $\tilde{v}$ を送るプロセス $E_x[\tilde{v}]$ と、復号化鍵 x^- を使って値を受けとり、チャネル x° に流すプロセス R_x を、以下のように定義します (F_x は補助的なプロセス) 。

$$\begin{aligned}E_x[\tilde{v}] \quad & \stackrel{\text{def}}{=} \quad \textbf{def key}\ k\ \textbf{in} \\ & \qquad \textbf{repl}\ \mathit{emit}\langle\{k\}_{x^+}\rangle \mid \\ & \qquad \textbf{let}\ k' = \textbf{filter}\ k\ \textbf{in repl}\ \mathit{emit}\langle\{\tilde{v}\}_{k'}\rangle \\ F_x \quad & \stackrel{\text{def}}{=} \quad \textbf{let}\ k = \textbf{filter}\ x^-\ \textbf{in} \\ & \qquad \textbf{def key}\ k'\ \textbf{in} \\ & \qquad\quad \textbf{repl}\ \mathit{emit}\langle\{k'\}_k\rangle \mid \\ & \qquad\quad \textbf{let}\ \tilde{y} = \textbf{filter}\ k'\ \textbf{in}\ x^\circ\langle\tilde{y}\rangle \\ R_x \quad & \stackrel{\text{def}}{=} \quad \textbf{repl}\ F_x\end{aligned}$$

このプロトコルだと、一旦送信が始まると永久に繰り返し続けるのでちょっと非現実的ですが、受信が成功したらちゃんと ack を返して送信を止めるようにすることも可能です。

A compositional translation

これに基づいて、 join-calculus のプロセス P を sjoin-calculus のプロセス $[\![P]\!]$ に対応させる変換 $[\![\cdot]\!]$ を、Figure 3 のように定義します。ここで $\mathbf{dv}[D]$ は D の中で定義されている名前や鍵の集合を、 $\bigwedge$ と $\prod$ は $\cdots \wedge \cdots \wedge \cdots$ や $\cdots \mid \cdots \mid \cdots$ を、それぞれ表します。

Soundness results

以上のように定義すると、 join calculus の任意のプロセス P, Q について、

$$P \approx Q \quad \Leftrightarrow \quad \mathit{Env}[[\![P]\!]] \approx \mathit{Env}[[\![Q]\!]]$$

が成り立ちます (full abstraction) 。 $\approx$ としては、いわゆる observational equivalence や、それより「粗い」任意の congruence (may-testing equivalence など) をとることができます。

証明は非常に複雑なので割愛させてもらいますが、プロトコルについて成り立ついくつかの抽象的な条件のみに基づいているので、それらを満たしていればこれ以外の実装でも同様の結果が得られます。たとえば、論文の full

version（現時点ではまだ draft ですが）では、必ずしもすべての通信を暗号化しない（「同じサイト」内の通信は暗号化しない）ような変換方法についても述べています。

5 おわりに

最近 Windows NT の仮想プライベート・ネットワークの実装（というかそもそもの仕様）が欠陥だらけであることが露呈して話題になっていますが、こういう formal method と現実の実装との gap がいつになっても埋まらないのは、何だかもどかしいなあと思いました。[4]

参考文献

[1] Martín Abadi. Protection in programming-language trnaslations. Technical report, Systems Research Center, 1998. Available from `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-154.html`.

[2] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science*, 1998. To appear.

[3] Raymie Stata and Martín Abadi. A type system for java bytecode subroutines. In *Proceedings of the 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[4] P.S. レジュメをゆっくり書く暇がなくて、全体的にかなり乱れた文章になってしまいました。どうもすみません。