Security for Multithreaded Programs under Cooperative Scheduling

Alejandro Russo and Andrei Sabelfeld

Dept. of Computer Science and Engineering, Chalmers University of Technology
412 96 Göteborg, Sweden, Fax: +46 31 772 3663

Abstract. Information flow exhibited by multithreaded programs is subtle because the attacker may exploit scheduler properties when deducing secret information from publicly observable outputs. Volpano and Smith have introduced a `protect` command that prevents the scheduler from observing sensitive timing behavior of protected commands and therefore prevents undesired information flows. While a useful construct, `protect` is nonstandard and difficult to implement. This paper presents a transformation that eliminates the need for `protect` under cooperative scheduling. We show that both termination-insensitive and termination-sensitive security can be enforced by variants of the transformation in a language with dynamic thread creation.

1 Introduction

Information-flow security specifications and enforcement mechanisms for sequential programs have been developed for several years. Unfortunately, they do not naturally generalize to multithreaded programs [17]. Information flow in multithreaded programs remains an important open challenge [12]. Furthermore, otherwise significant efforts (such as Jif [7] and Flow Caml [14]) in extending programming languages (such as Java and Caml) with information flow controls have sidestepped multithreading issues. Nevertheless, concurrency and multithreading are important in the context of security because environments of mutual distrust are often concurrent. As result, the need for controlling information flow in multithreaded programs has become a necessity.

This paper is focused on preventing attacks that exploit scheduler properties to deduce secret information from publicly observable outputs. Suppose h is a secret (or *high*) variable and l is a public (or *low*) one. Consider threads c_1 and c_2:

$$c_1 : (\texttt{if}\ h > 0\ \texttt{then}\ \texttt{sleep}(100)\ \texttt{else}\ \texttt{skip});\ l := 1$$
$$c_2 : \texttt{sleep}(50);\ l := 0$$

Although these threads do not exhibit insecure information flow in isolation (because 1 is always the outcome for l in c_1, and 0 is always the outcome for l in c_2), there is a race between assignments $l := 1$ and $l := 0$, whose outcome depends on secret h. If h is originally positive, then—under many schedulers—it is likely that the final value of l is 1. If h is not positive, then it is likely that the final value of l is 0. It is the timing behavior of thread c_1 that leaks—via the scheduler—secret information into l. This phenomenon is due to *internal timing*, i.e., timing that is observable to the scheduler.

$$\frac{\langle\!| c_i, m |\!\rangle \xrightarrow{\alpha} \langle\!| c_i', m' |\!\rangle \qquad \alpha \in \{\epsilon, \vec{d}\} \qquad \sigma = i}{\langle\!| \sigma, \langle c_1 \dots c_n \rangle, m |\!\rangle \rightarrow \langle\!| \sigma, \langle c_1 \dots c_{i-1} c_i' \alpha c_{i+1} \dots c_n \rangle, m' |\!\rangle}$$

$$\frac{\langle\!| c_i, m |\!\rangle \xrightarrow{\alpha} \langle\!| \mathit{stop}, m' |\!\rangle \qquad \sigma = i}{\langle\!| \sigma, \langle c_1 \dots c_n \rangle, m |\!\rangle \rightarrow \langle\!| \sigma, \langle c_1 \dots c_{i-1} c_{i+1} \dots c_n \rangle, m' |\!\rangle}$$

$$\frac{\langle\!| c_i, m |\!\rangle \xrightarrow{\not\curvearrowright} \langle\!| c_i', m |\!\rangle \qquad \sigma = i \qquad \sigma' = (i \bmod n) + 1 \qquad c_i' \neq \mathit{stop}}{\langle\!| \sigma, \langle c_1 \dots c_n \rangle, m |\!\rangle \rightarrow \langle\!| \sigma', \langle c_1 \dots c_{i-1} c_i' c_{i+1} \dots c_n \rangle, m |\!\rangle}$$

Fig. 1. Semantics for threadpools

As in [17, 18, 15, 1, 16, 8], we do not consider *external timing*, i.e., timing behavior visible to an attacker with a stopwatch.

Volpano and Smith have introduced a `protect` command that prevents the scheduler from observing the timing behavior of the protected command and therefore prevents undesired information flows. A protected command is executed atomically *by definition*. Although it has been acknowledged [13, 8] that `protect` is hard to implement, no implementation of `protect` has been discussed by approaches that rely on it [18, 15, 16]. This paper presents a transformation that eliminates the need for `protect` under cooperative scheduling. This transformation can be integrated into source-to-source translation that introduces `yield` commands for cooperative schedulers. We show that both termination-insensitive and termination-sensitive security can be enforced by variants of the transformation in a language with dynamic thread creation.

2 Language

We consider a simple imperative language that includes `skip`, assignment, sequential composition, conditionals, and `while`-loops. Its sequential semantics is standard [20]. The language also includes dynamic thread creation and a `yield` command. A *command configuration* $\langle\!| c, m |\!\rangle$ consists of a command c and memory m. Memories $m : \mathit{IDs} \rightarrow \mathit{Vals}$ are finite maps from identifier names *IDs* to values *Vals*. Transitions between configurations have form $\langle\!| c, m |\!\rangle \xrightarrow{\alpha} \langle\!| c', m' |\!\rangle$ where α is either ϵ (empty label), $\vec{d}$ (indicating a sequence of newly spawned threads), or $\not\curvearrowright$. The latter label is used in the transition rule for `yield`:

$$\langle\!| \texttt{yield}, m |\!\rangle \xrightarrow{\not\curvearrowright} \langle\!| \mathit{stop}, m |\!\rangle$$

Labels are propagated through sequential composition to the threadpool-semantics level. Dynamic thread creation is performed by command `fork`:

$$\langle\!| \texttt{fork}(c, \vec{d}), m |\!\rangle \xrightarrow{\vec{d}} \langle\!| c, m |\!\rangle$$

This has the effect of continuing with thread c while spawning a sequence of fresh threads $\vec{d}$. *Threadpool configurations* have form $\langle\!| \sigma, \langle c_1 \dots c_n \rangle, m |\!\rangle$ where σ is the scheduler's running thread number, $\langle c_1 \dots c_n \rangle$ is a threadpool, and m is a shared memory.

Threadpool semantics, describing the behavior of threadpools and their interaction with the scheduler, are displayed in Figure 1. The rules correspond to normal execution of thread i from the threadpool, termination of thread i, and yielding by thread i. Note that due to cooperative scheduling, only termination or a `yield` by a thread may change the decision of the scheduler which thread to run next. Although these semantics model a round-robin scheduler, our approach can be generalized to a wide class of schedulers.

Let $cfg \to^0 cfg$, for any configuration cfg, and $cfg \to^v cfg'$, for $v > 0$, if there is a configuration cfg'' such that $cfg \to cfg''$ and $cfg'' \to^{v-1} cfg'$. Then, $cfg \to^* cfg'$ if $cfg \to^v cfg'$ for some $v \geq 0$. Threadpool configuration cfg *terminates* in memory m (written $cfg \Downarrow m$) if $cfg \to^* \langle\!| \sigma, \langle\rangle, m |\!\rangle$ for some σ. In particular, $cfg \Downarrow^v m$ is written when $cfg \to^v \langle\!| \sigma, \langle\rangle, m |\!\rangle$. If $\langle\rangle$ is not finitely reachable from cfg, then cfg *diverges* (written $cfg \Uparrow$). Termination $\Downarrow$ and divergence $\Uparrow$ are defined similarly for command configurations.

3 Security specification

We define two security conditions, termination-insensitive and termination-sensitive security, both based on *noninterference* [4]. Suppose *security environment* $\Gamma : IDs \to \{high, low\}$ specifies a partitioning of variables into high and low ones. Two memories m_1 and m_2 are *low-equal* ($m_1 =_L m_2$) if they agree on low variables, i.e., $\forall x \in IDs.\, \Gamma(x) = low \Longrightarrow m_1(x) = m_2(x)$.

Command c satisfies termination-insensitive noninterference if c's terminating executions on low-equal inputs produce low-equal results.

Definition 1. *Command c satisfies* termination-insensitive security *if*

$$\forall m_1, m_2. m_1 =_L m_2 \;\;\&\;\; \langle\!| 1, \langle c\rangle, m_1 |\!\rangle \Downarrow m_1' \;\;\&\;\; \langle\!| 1, \langle c\rangle, m_2 |\!\rangle \Downarrow m_2' \Longrightarrow m_1' =_L m_2'$$

Command c satisfies termination-sensitive noninterference if c's executions on any two low-equal inputs either both diverge or both terminate in low-equal results.

Definition 2. *Command c satisfies* termination-sensitive security *if*

$$\forall m_1, m_2. m_1 =_L m_2 \Longrightarrow$$
$$\langle\!| 1, \langle c\rangle, m_1 |\!\rangle \Downarrow m_1' \;\&\; \langle\!| 1, \langle c\rangle, m_2 |\!\rangle \Downarrow m_2' \;\&\; m_1' =_L m_2' \vee \langle\!| 1, \langle c\rangle, m_1 |\!\rangle \Uparrow \& \; \langle\!| 1, \langle c\rangle, m_2 |\!\rangle \Uparrow$$

4 Transformation

By performing a simple analysis while injecting `yield` commands, we are able to automatically enforce both termination-insensitive and termination-sensitive security. The transformation rules are presented in Figure 2. They have form $\Gamma \vdash c \hookrightarrow c'$, where command c is transformed into c' under Γ. In order to rule out *explicit flows* [2] via assignment, we ensure that expressions assigned to low variables may not depend on high data. This is enforced by demanding the type of the assigned variable to be at least as restrictive as the type of the expression that is to be assigned. Restrictiveness relation $\sqsubseteq$ on security levels is defined by $low \sqsubseteq low$, $high \sqsubseteq high$, $low \sqsubseteq high$ and $high \not\sqsubseteq low$.

$$\frac{\forall v \in Vars(e).\, \Gamma(v) = low}{\Gamma \vdash e : low} \qquad \frac{\exists v \in Vars(e).\, \Gamma(v) = high}{\Gamma \vdash e : high}$$

$$\text{(HCTX)}\frac{\text{No } \texttt{yield}, \texttt{fork} \text{ or assignment to } l \text{ in } c}{\Gamma \vdash c : high}$$

$$\frac{}{\Gamma \vdash \texttt{skip} \hookrightarrow \texttt{skip}; \texttt{yield}} \qquad \frac{}{\Gamma \vdash \texttt{yield} \hookrightarrow \texttt{yield}}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \sqsubseteq \Gamma(v)}{\Gamma \vdash v := e \hookrightarrow v := e; \texttt{yield}} \qquad \frac{\Gamma \vdash c_1 \hookrightarrow c_1' \quad \Gamma \vdash c_2 \hookrightarrow c_2'}{\Gamma \vdash c_1; c_2 \hookrightarrow c_1'; c_2'}$$

$$\frac{\Gamma \vdash e : low \quad \Gamma \vdash c_1 \hookrightarrow c_1' \quad \Gamma \vdash c_2 \hookrightarrow c_2'}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow \texttt{if}\ e\ \texttt{then}\ (\texttt{yield}; c_1')\ \texttt{else}\ (\texttt{yield}; c_2')}$$

$$\text{(H-IF)}\frac{\Gamma \vdash e : high \quad \Gamma \vdash c_1 : high \quad \Gamma \vdash c_2 : high}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2); \texttt{yield}}$$

$$\frac{\Gamma \vdash e : low \quad \Gamma \vdash c \hookrightarrow c'}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow (\texttt{while}\ e\ \texttt{do}\ (\texttt{yield}; c')); \texttt{yield}}$$

$$\text{(H-W)}\frac{\Gamma \vdash e : high \quad \Gamma \vdash c : high}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow (\texttt{while}\ e\ \texttt{do}\ c); \texttt{yield}}$$

$$\frac{\Gamma \vdash c \hookrightarrow c' \quad \Gamma \vdash d_1 \hookrightarrow d_1' \quad \ldots \quad \Gamma \vdash d_n \hookrightarrow d_n'}{\Gamma \vdash \texttt{fork}(c, d_1 \ldots d_n) \hookrightarrow \texttt{fork}(c', d_1' \ldots d_n')}$$

Fig. 2. Transformation rules

In order to reject *implicit flows* [2] via control flow, we guarantee that `if`'s and `while`'s with high guards may not have assignments to low variables in their bodies. These two techniques are well known [2, 19] and do not require code transformation.

The transformation injects `yield` commands in such a way that threads may not yield whenever their timing information depends on secret data. This is achieved by a requirement that `if`'s and `while`'s with high guards may not contain `yield` commands. In addition, such control flow statements may not contain `fork`. The rationale is that if secrets influence the number of threads, then it is possible for some schedulers to leak this difference via races of publicly-observable assignments [13, 10]. Rules H-IF and H-W enforce the above requirements. The rest of the transformation injects `yield` commands without significant restrictions (but with some obvious liveness guarantees for commands that do not branch on secrets).

The first lemma shows that commands typed under rule HCTX do not affect the low-security variables.

Lemma 1. *Given a command c and memories m_1 and m_2 so that $\Gamma \vdash c : high$, $m_1 =_L m_2$, $\langle\!| c, m_1 |\!\rangle \Downarrow m_1'$, and $\langle\!| c, m_2 |\!\rangle \Downarrow m_2'$, then $m_1' =_L m_2'$.*

The following theorem states that pools of transformed threads preserve low-equality on memories:

Theorem 1. *Given two (possibly empty) threadpools $\vec{c}$ and $\vec{c}\,'$ of equal size, memories m_1 and m_2, and number σ so that $\Gamma \vdash c_i \hookrightarrow c_i'$ where $c_i \in \vec{c}$ and $c_i' \in \vec{c}\,'$, $m_1 =_L m_2$, $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_1 |\!\rangle \Downarrow^v m_1'$, and $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_2 |\!\rangle \Downarrow^w m_2'$, then $m_1' =_L m_2'$.*

Proof. The proof is done by induction on $v + w$. □

As desired, the transformation enforces termination-insensitive security:

Corollary 1. *If $\Gamma \vdash c \hookrightarrow c'$ then c' satisfies termination-insensitive security.*

Proof. By applying Theorem 1 with $\vec{c} = \langle c \rangle$, $\vec{c}\,' = \langle c' \rangle$, and $\sigma = 1$. □

The transformation can be adopted to termination-sensitive security in a straightforward way. We write $\Gamma \vdash_{\mathrm{TS}} c \hookrightarrow c'$ whenever $\Gamma \vdash c \hookrightarrow c'$ with the modifications that (i) rule H-W is not used, and (ii) rule HCTX is replaced by:

$$\text{(HCTX')}\ \frac{\text{No } \texttt{while}, \texttt{yield}, \texttt{fork} \text{ or assignment to } l \text{ in } c}{\Gamma \vdash_{\mathrm{TS}} c : \mathit{high}}$$

These modifications ensure that loops have low guards and that no loop may appear in an `if` statement with a high guard. These requirements are similar to those of Volpano and Smith [18] (except for the requirement on `fork`, which Volpano and Smith lack):

Lemma 2. *Given a command c so that $\Gamma \vdash c : high\ cmd$ for some security environment Γ in Volpano and Smith's type system [18]; and given command c' obtained from c by erasing occurrences of* `protect`*, we have $\Gamma \vdash_{TS} c' : high$.*

Proof. By structural induction on the type derivation of c. □

This allows us to connect the transformation to Volpano and Smith's type system:

Theorem 2. *If command c is typable under security environment Γ in Volpano and Smith's type system [18], then there exists command c'' such that $\Gamma \vdash_{TS} c' \hookrightarrow c''$, where c' is obtained from c by erasing occurrences of* `protect`.

Proof. By structural induction on the type derivation of c and Lemma 2. □

We also achieve termination-sensitive security with the above modifications of the transformation. We firstly present some auxiliaries lemmas. The following lemma states that commands typed as *high* terminate and do not affect the low part of the memory:

Lemma 3. *Given a command c and memory m so that $\Gamma \vdash_{TS} c : high$, then $\langle\!| c, m |\!\rangle \Downarrow m'$ and $m =_L m'$.*

Proof. By induction on the size of c. □

In order to show termination-sensitive security, we track the behavior of threadpools after executing some number of `yield` and `fork` commands. We capture this by relation $\rightarrow^*_{y,f}$ so that $cfg \rightarrow^*_{1,0} cfg'$ if there is cfg'' such that $cfg \rightarrow^* cfg''$ where no `yield`'s have been executed, $cfg'' \rightarrow cfg'$ results from executing a `yield` command; and $cfg \rightarrow^*_{y,f} cfg'$ if there is cfg'' such that $cfg \rightarrow^*_{y-1,f} cfg''$ (resp. $cfg \rightarrow^*_{y,f-1} cfg''$) and $cfg'' \rightarrow cfg'$ results from executing a `yield` (resp. `fork`) command.

The next two lemmas state that low-equivalence between memories is preserved after executing some number of `yield` and `fork` commands:

Lemma 4. *Given two non-empty threadpools $\vec{c}$ and $\vec{c}\,'$ of equal size, memories m_1 and m_2, and number σ so that $\Gamma \vdash_{TS} c_i \hookrightarrow c_i'$ where $c_i \in \vec{c}$ and $c_i' \in \vec{c}\,'$, $m_1 =_L m_2$, and $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_1 |\!\rangle \rightarrow^*_{1,0} \langle\!| \sigma', \langle \vec{c}\,'' \rangle, m_1' |\!\rangle$, then there exists m_2' such that $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_2 |\!\rangle \rightarrow^*_{1,0} \langle\!| \sigma', \langle \vec{c}\,'' \rangle, m_2' |\!\rangle$, and $m_1' =_L m_2'$.*

Proof. By simple induction on the number of steps of $\rightarrow^*_{1,0}$. $\square$

Lemma 5. *(`yield`/`fork` lock-step execution) Given two non-empty threadpools $\vec{c}$ and $\vec{c}\,'$ of equal size, memories m_1 and m_2, numbers σ, y, and f so that $\Gamma \vdash_{TS} c_i \hookrightarrow c_i'$ where $c_i \in \vec{c}$ and $c_i' \in \vec{c}\,'$, $m_1 =_L m_2$, and $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_1 |\!\rangle \rightarrow^*_{y,f} \langle\!| \sigma', \langle \vec{c}\,'' \rangle, m_1' |\!\rangle$, then there exists m_2' such that $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_2 |\!\rangle \rightarrow^*_{y,f} \langle\!| \sigma', \langle \vec{c}\,'' \rangle, m_2' |\!\rangle$, and $m_1' =_L m_2'$.*

Proof. By induction on $y + f$ and by applying Lemmas 3 and 4 when necessary. $\square$

The final theorem shows that the transformation eliminates the need for `protect`:

Theorem 3. *If $\Gamma \vdash_{TS} c \hookrightarrow c'$ then c' satisfies termination-sensitive security.*

Proof. By applying Lemma 5 with $\vec{c} = \langle c \rangle$, $\vec{c}\,' = \langle c' \rangle$, and $\sigma = 1$ and observing that a divergent configuration (originating from c') performs an infinite number of `yield`'s. $\square$

5 Related work

An overview of information flow controls for concurrent programs can be found in [12]. We briefly mention most closely related work. External timing-sensitive information-flow policies have been addressed for a multithreaded language [13], and extended with synchronization [9], message passing [11], and declassification [6]. Type systems have been investigated for termination-sensitive flows in possibilistic [1] and probabilistic [18, 15, 16] settings. Recently, we have presented a type system that guarantees termination-insensitive security with respect to a class of deterministic schedulers [8]. Information flow via low determinism, prohibiting races on low variables from the outset, has been addressed in [21, 5].

6 Conclusion

We have presented a transformation that prevents timing leaks via cooperative schedulers. We argue that this technique is general: it applies to a wide class of schedulers (although only a round-robin scheduler has been considered here for simplicity).

We have experimented with the GNU Pth [3], a portable thread library for threads in user space. We have modified this library to allow the round-robin scheduling policy from Section 2. We have successfully applied the transformation for source-to-source translation of multithreaded programs without `yield`'s into GNU Pth programs. The security of this translation is ensured by Theorems 1 and 3.

Acknowledgment This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 Mobius project.

References

[1] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.

[2] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[3] R. S. Engelschall. Gnu pth - the gnu portable threads. `http://www.gnu.org/software/pth/`, Nov. 2005.

[4] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[5] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Security Foundations Workshop*, July 2006.

[6] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, Nov. 2004.

[7] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. `http://www.cs.cornell.edu/jif`, July 2001–2006.

[8] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.

[9] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[10] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2890 of *LNCS*, pages 260–273. Springer-Verlag, July 2003.

[11] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, Sept. 2002.

[12] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[13] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[14] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[15] G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[16] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Security Foundations Workshop*, pages 3–13, 2003.

[17] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[18] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, Nov. 1999.

[19] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[20] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.

[21] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.