Eliminating Timing Leaks by Unification (Extended Abstract)

Boris Köpf and Heiko Mantel

ETH Zürich, Switzerland
{boris.koepf,heiko.mantel}@inf.ethz.ch

Abstract. Transforming security type systems [Aga00] go beyond checking whether a given program has secure information flow. Rather than simply rejecting a program with insecure information flow, they construct a program that has secure information flow and whose behavior is similar enough to that of the original program such that it can act as a replacement. In this extended abstract, we sketch ongoing work on improving transforming type systems by incorporating unification.

1 Introduction

Static analysis of information flow security on the level of concrete programming languages has received much attention in recent years with advances in theory and practice [SM03]. Security type systems provide a basis for reasoning in a modular fashion, guided by the program structure, about the information flow within a program. Each rule of such a type system performs a local analysis of the programs top-level constructor and, if it is applicable, either determines directly that the program is secure (for primitive commands) or reduces the analysis of the program to the analysis of its subprograms (for composed commands). The type check fails if no rule is applicable to the top-level constructor.

Such a purely local approach to reasoning about secure information flow has obvious advantages in terms of simplicity and efficiency. Its precision, however, is fairly limited as it cannot cope very well with global aspects of information flow such as *implicit flow* [Den82], i.e. information flow due to the flow of control. For instance, if the value of the guard in an if-then-else statement depends on a secret then one must make sure that an untrusted user cannot tell from his observations which branch has been taken because, otherwise, he would learn information about the secret. That is, the two branches must be observationally equivalent from the perspective of any untrusted observer, which is not a question of a single constructor. While there obviously is a safe approximation, namely to simply make the type check fail if a secret variable appears in the guarding expression, this is a rather severe restriction. More sophisticated typing rules employ global side conditions that result in slightly better approximations such as, e.g., if the guard contains a secret variable then no assignments to variables that can be observed by untrusted users are permitted in the branches. In a setting with multi-threading, one must additionally ensure that the value of the

secret variable in the guard has no influence on the timing behavior. This can be achieved by executing conditionals, including the commands in the chosen branch, atomically [VS98] or by ensuring that both branches have identical run time [SS00]. The current solutions are, however, still fairly simplistic and do not add as much to the precision of the analysis as one would wish.

In this extended abstract, we propose a more balanced approach to combining local reasoning about individual commands with global reasoning about entire sub-programs. Technically, we employ *security type systems for local reasoning* and *unification for global reasoning*. We illustrate this approach by extending an existing analysis technique, namely the one from [SS00].

2 Transforming Security Type Systems

We assume a security policy with two security domains, *high* and *low*, where information flow from *high* to *low* is forbidden. Given an imperative program, each variable is associated with one of these two security domains with the intuition that initial values of high variables are the secrets that shall not be leaked and that attackers can only observe values of low variables. That is, a program has *secure information flow* if the initial values of *high* variables do not affect the values of *low* variables during program execution. This is the usual setting for investigating information flow security on the level of concrete imperative programming languages.

Security type systems provide a mechanism for automatically checking whether a given program has secure information flow. Being typable means for a program that it has secure information flow. It has become common practice that security type systems are proved to be sound with respect to a more abstract formal definition of secure information flow. Completeness is sacrificed for making an efficient automation of the security analysis possible. That is, if a program is not typable then this does not necessarily mean that the program has insecure information flow. However, the analysis should reject as few secure programs as possible, i.e. it should be precise.

Transforming type systems go beyond checking whether a given program has secure information flow. Rather than simply rejecting a program that is insecure, they modify it such that the resulting program is secure. This can be captured by a judgment of the form $C \hookrightarrow C' : S$ (meaning *the program C is transformed into a secure program C' with type S"*).

Any transformation that is capable of making insecure programs secure necessarily also changes the program's behavior in some way. There is a natural trade-off between the changes to the program's behavior that one is willing to accept and the information leaks that can be corrected by the transformation. Here, we restrict transformations to ones that affect the timing of a given program but no other aspects of its behavior. That is, the transformed program shall be a slowed-down version of the original program. The information leaks that can be eliminated under this restriction are so called *timing leaks*, which can occur if a program's run time depends on the values of secret variables. Ob-

viously, timing leaks can be exploited if the attacker is capable of measuring the run time. Moreover, timing leaks can be exploited in a multi-threaded setting even if attackers do not have access to precise stop watches [SS00].

Timing leaks occur in a conditional if the guard depends on the value of a *high* variable and the two branches have different run times. Agat suggested a transformation that eliminates timing leaks by *padding*. In this approach, the type S of a program P is itself also a program and S has the same observational behavior as P from the perspective of a *low* user. That is, for any two starting states s, t that look the same to a *low* observer, the runs of P and S starting in s and t, respectively, have the same length, i.e. identical run time, and the corresponding states in the two runs look the same. Hence, timing leaks can be eliminated by sequentially composing each branch of a conditional with the type of the respective other branch. This holds in a sequential and also in a multi-threaded setting [SS00]. The transformation preserves all aspects of a program's behavior except for timing, which is due to the construction of types (types contain no assignments to *high* variables) and to a side condition in the typing rule for conditionals with *high* guards, which is viewed below (no assignments to low variables in the types of the branches, $al(S_1) = al(S_2) = false$).

$$\frac{B : high \quad C_1 \hookrightarrow C_1' : S_1 \quad C_2 \hookrightarrow C_2' : S_2 \quad al(S_1) = al(S_2) = false}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ \{C_1'; S_2\}\ \mathsf{else}\ \{S_1; C_2'\} : \mathsf{skip}; S_1; S_2}$$

3 Current Limitations

The transforming type system sketched in the previous section can be used to check whether a given program has secure information flow and the transformation eliminates timing leaks to some extent, but it is not yet an optimal solution:

1. The program `if h≤0 then h:=h+1 else h:=h-1` is strongly secure.[1] However, if we apply the typing rule from the previous section then this results in the program `if h≤0 then {h:=h+1;skip} else {skip;h:=h-1}`. Applying the typing rule again results in `if h≤0 then {h:=h+1;skip;skip;skip} else {skip;skip;skip;h:h-1}`. In summary, the transformation not only eliminates timing leaks from insecure programs but also modifies secure programs that are secure already. Moreover, the transformation is not idempotent, which also shows that the transformation modifies some secure programs. Interestingly, the size of a program grows, in the worst case, exponentially in the number of applications of the transformation.
2. The program `if h≤0 then l:=l+1 else l:=l+1` is strongly secure. However, the typing rule from the previous section is not applicable for this program as the branches contain assignments to *low* variables, i.e. the type check fails. For the same reason, programs like `if h≤0 then {h:=h+1;l:=l+1} else {l:=l+1;h:=h-1}` cannot be corrected by the transformation although there is an obvious correction: `if h≤0 then {h:=h+1;l:=l+1;skip} else {skip;l:=l+1;h:=h-1}`.

[1] Strong security [SS00] is the semantic security definition underlying Sabelfeld and Sands's security type system.

3. The side condition in the typing rule for conditionals with *high* guards (no assignments to low variables) is somewhat incompatible with the typing rule for while loops (depicted below). The latter rule requires that the guard of a while loop contains no other variables than *low* variables. That is, while loops occurring in the branch of a conditional with a high guard either terminate immediately (guard evaluates to false when the while loop is reached) or do not terminate (assignments to low variables are forbidden in the branches of a conditional with a high guard). Hence, if the guard evaluates to true when the loop is reached it will remain true. This severely limits the use of loops.

$$\frac{B : \mathit{low} \quad C \hookrightarrow C' : S}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ S}$$

4 Incorporating Unification

The branches of a conditional with a *high* guard must be observationally equivalent for a *low* observer. That is, in the two branches, the same values must be assigned to any given *low* variable, assignments to *low* variables must occur in the same order and at the same point of time, and the overall run time of the branches must be identical. The transformation is limited to the insertion of skip statements as it may slow down the program but must not make any other changes to the program. To this end, we massage each of the branches by inserting meta variables that may then be substituted by the sequential composition of an arbitrary number of skip statements (including the sequential composition of zero skip statements, denoted by ϵ). For instance, the program if h≤0 then {h:=h+1;l:=l+1} else {l:=l+1;h:=h-1} is lifted to the program if h≤0 then {h:=h+1;α_1;l:=l+1;α_2} else {α_3;l:=l+1;h:=h-1;α_4}. One obtains a secure program by applying the substitution $\alpha_1\backslash\epsilon$, $\alpha_2\backslash$skip, $\alpha_3\backslash$skip, and $\alpha_4\backslash\epsilon$. We employ unification for automatically computing these substitutions. As the substitutions need not make the two programs syntactically identical but rather only observationally equivalent, we unify under a given equational theory that incorporates equivalences like, e.g., $h := \mathit{Exp} \mathrel{\hat{=}}_L$ skip (skip is observationally equivalent for a *low* observer to assignments to *high* variables), $h := \mathit{Exp} \mathrel{\hat{=}}_L h' := \mathit{Exp}'$ (two high assignments are observationally equivalent), and $l := \mathit{Exp} \mathrel{\hat{=}}_L l := \mathit{Exp}'$ for expressions *Exp* and *Exp*′ that evaluate to identical values in identical states (semantically identical assignments to low variables are observationally equivalent). Moreover, e.g., if B then C_1 else $C_2 \mathrel{\hat{=}}_L$ skip; C_3 if $C_1 \mathrel{\hat{=}}_L C_3$ and $C_2 \mathrel{\hat{=}}_L C_3$ (the evaluation of a guard is equivalent to a skip statement). We denote the set of all unifiers for two programs C_1 and C_2 under the equational theory by $\mathcal{U}_{\mathrel{\hat{=}}_L}(C_1, C_2)$ and the application of a substitution σ to a program C by $C\sigma$. This leads to the following transforming typing rule for conditionals with *high* guards:

$$\frac{\begin{array}{c}\mathit{dom}(B) = \mathit{high} \quad C_1 \hookrightarrow C_1' : S_1 \quad C_2 \hookrightarrow C_2' : S_2 \\ \overline{C_1'} : \overline{S_1} = \mathit{lift}(C_1' : S_1) \quad \overline{C_2'} : \overline{S_2} = \mathit{lift}(C_1' : S_2) \quad \sigma \in \mathcal{U}_{\mathrel{\hat{=}}_L}(\overline{S_1}, \overline{S_2})\end{array}}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ \overline{C_1'}\sigma\ \mathsf{else}\ \overline{C_2'}\sigma : \mathsf{skip}; \overline{S_1}\sigma}$$

We are currently in the process of adopting known unification algorithms to efficiently compute the set $\mathcal{U}_{\triangleq_L}(S_1, S_2)$ of unifiers.

We see three main advantages of our approach.

Firstly, the precision of type checking is improved. Programs are not simply rejected because branches of conditionals with *high* guards incorporate assignments to *low* variables, but are carefully inspected. For instance, the program `if h≤0 then l:=l+1 else l:=l+1` is rejected by the original type system despite it has secure information flow. Our new type system correctly accepts this program.

Secondly, the transformation is capable of correcting more timing leaks. For instance, the program `if h≤0 then {h:=h+1;l:=l+1} else {l:=l+1;h:=h-1}` is rejected by the original transformation and could not be corrected. Our new type system transforms it into `if h≤0 then {h:=h+1;l:=l+1;skip} else {skip;l:=l+1;h:=h-1}`, which is a secure program. Being able to type check and to transform programs where while loops occur in branches of conditionals with a high guard, also broadens the applicability of the transforming type system considerably.

Thirdly, the transformation leads to more compact and more efficient programs. For instance, the program `if h≤0 then h:=h+1 else h:=h-1` type checks without any need for transformations. This is unlike in the original type system. More generally, our transformation is idempotent.

Initial results indicate that we can obtain all of these improvements without having to give up any of the desirable features of the original type system, including soundness and the fact that the transformation affects no other aspects of a program's behavior than its timing. Elaborating this in full detail is the aim of on-going work.

References

[Aga00] J. Agat. Transforming out Timing Leaks. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 40–53, 2000.

[Den82] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[SM03] A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.

[SS00] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–215, Cambridge, UK, 2000.

[VS98] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 34–43, Rockport, Massachusetts, 1998.