Secure Self-Certified COTS*

M. Debbabi & E. Giasson & B. Ktari & F. Michaud & N. Tawbi
LSFM Research Group,
Computer Science Department,
Science and Engineering Faculty,
Laval University,
Quebec, Canada
{debabi,giassone,ktari,michaudf,tawbi}@ift.ulaval.ca

Abstract

With the advent and the rising popularity of networks, Internet, intranets and distributed systems, security is becoming one of the major concerns in IT research. An increasing number of approaches have been proposed to ensure the safety and security of programs. Among those approaches, certified code seems to be the most promising. Unfortunately, as of today, most of the research on certified code have focused on simple type safety and memory safety, rather than security issues. We therefore propose to extend this approach to the security aspects of a program. Our intention is to use such an approach as an efficient and realistic solution to the problem of malicious code detection in COTS. In this paper, we present our progress in defining and implementing a certifying compiler that produces a secure self-certified code that can be used to ensure both safety and security of the code.

1. Motivation and Background

Nowadays, there are many information infrastructures based on the so-called Commercial Off-The-Shelf (COTS) components. Actually, many organizations are undergoing a remarkable move from legacy systems towards COTS-based systems. The main motivation underlying such a migration is to take advantage of cutting-edge technologies and also to lower the program life-cycle costs of computer systems. Nevertheless, this migration phenomenon poses major and yet very interesting challenges to the currently established computer system technologies in terms of security, reliability, integration, interoperability, maintenance, planning, etc.

*This research is funded by a research contract from the Defense Research Establishment, Valcartier, DREV, Quebec, Canada.

In our current research, we are concerned with malicious code that could exist in COTS software products. In a preliminary study of the domain [2], we considered different approaches suitable to address this problem. Among those approaches, certified code seems to be the most promising. Given a certified program provided by an untrusted source, a host can determine with certainty that this program may be safely executed. Unfortunately, as of today, most of the research on certified code have focused on simple type safety and memory safety, rather than security issues. We therefore propose to extend this approach to the security aspects of a program.

In this paper, we present our progress in defining and implementing a certifying compiler that produces a secure self-certified code that can be used to ensure both the safety and security of the code. We used LCC [3], a C compiler, as the starting point for our certifying compiler to which we added a type system based on TALx86 [5].

The rest of the paper is organized as follows. Section 2 presents the certified code approach, which guarantees that COTS components may be safely executed. Section 3 focuses on the code generation process that produces annotated assembly code. Section 4 follows with the verification process, which aims at formally certifying that the generated annotated code satisfies well defined safety properties. Section 5 proposes an extension to both the generation and verification processes to include security properties. Section 6 presents related work. Finally, a few concluding remarks and a discussion of future research are ultimately sketched as a conclusion in Section 7.

2. Certified Code

Certified code is a general framework that aims at guaranteeing COTS components compliance with the functionality and security requirements. It allows a host, which is re-

ferred to as the *code consumer*, to determine with certainty that a program supplied by an untrusted source, which is referred to as the *code producer*, can be safely executed. The code producer must attach a collection of annotations (the *certificate*) to the code it produces. The certificate can be made of proofs, types, or any other kind of formal or informal information about program dynamic behaviors properties. Once the code consumer holds both the code and the certificate, it checks the certificate to ensure that it is safe to execute the associated code.

To its full extent, the certified code approach brings several key characteristics that draw a clear advantage over more traditional approaches for malicious code detection:

- First, the overall certification process is shared between the producer and the consumer code. Thus, as the code verification is much easier than the code and certificate generation, this could be beneficial for the consumer compared to other approaches, such as a full blown, lengthy static binary code analysis.
- Second, approaches using cryptography and authentication, in order to ensure that the code has been produced by a trusted source and has not been altered during the transmission, are not required. If either the code or the certificate is altered, the verification process will fail. Moreover, if both are altered in such a manner that the verification still succeeds, then the code remains safe, provided that the certificate still satisfies safety and security properties.
- Third, since a certificate is attached to the low-level code and the former contains additional information regarding the code (proofs, types, etc.), the verification process could take advantage of this additional information, as opposed to a raw binary code verification.
- Lastly, the verification process is performed off-line and only once, no matter how many times the code is executed. Thus, there are no run-time penalties, which makes this approach more efficient than others.

The certified code approach first appeared in the JAVA programming language. The JAVA compiler produces statically typed Java Virtual Machine Language (JVML) code, which includes typing annotations that a verifier can use to ensure the type safety of the code. More recently, three interesting and more general approaches have been proposed:

- Proof-Carrying Code (PCC) [7]: The certificate represents a proof. The producer supplies along with the code a proof attesting that the code satisfies a formally defined security policy publicized by the consumer. The consumer then gets the code and its proof, and validates the proof in order to execute the code safely.
- Typed Assembly Language (TAL) [6]: The certificate is made of typing annotations. Along with the assembly code, the compiler produces typing annotations that can be used to ensure the safety of the assembly code. Thus, the verification process consists of type-checking the annotated assembly code.
- Efficient Code Certification (ECC) [4]: The certificate corresponds to structured annotations that merely direct the verification process. Annotations are limited to a fixed and relatively basic level of security (control flow, memory and stack safety) since the motivation of the author was to propose a simple and efficient system that would be relatively simple to incorporate into existing compilers.

3. Code Generation

To implement our certifying compiler, we modified an existing compiler instead of building a new one from scratch. We chose LCC since its source code (about 15 000 lines) is available for free and the book [3] covers it nicely.

LCC is a retargetable compiler for ANSI C that generates code for the following processors: ALPHA, SPARC, MIPS R3000 and Intel x86. To allow compilation for different processors, LCC is designed in two parts: a general front end and a processor-dependent back end. The front end does the C language analysis and produces an intermediate representation of directed acyclic graphs. Afterwards, the back end uses this intermediate representation to generate the assembly code.

Taking back the idea of a typed assembly language, which is the TAL project (throughout this paper, the TAL acronym refers to this particular project, instead of “typed assembly language”, its actual meaning), the first step in our certified compilation process is to generate a typed assembly output from an ANSI C source file. Basically, typed assembly is an annotated assembly code. Annotations, created and inserted automatically by a compiler, describe the types of each data and piece of code in a given source file as well as types of variables and functions imported from other files or libraries. Types are built from a type algebra that allows a low-level description of both low-level concepts such as integers and hardware registers, and higher level concepts such as structures and functions. Such a type system must be closely related to both the source language (here ANSI C) and the destination platform hardware (32-bits code on an Intel x86). Indeed, types serve as the missing link between structured and functional source code and machine dependent assembly language.

Our type algebra was largely inspired by TAL type system, at least textually, though semantics were widely redefined and tailored to our needs. While TAL types were

based on the Popcorn language (a sort of a C subset and ML hybrid language), ours were meant to fully describe ANSI C elements, such as pointers, integers of different sizes, floating-point numbers, etc.

Both data and code labels need to be annotated, as well as instructions that branch to code labels, namely calls and jumps. On the data side, there exists an almost one to one relation between ANSI C data types and low-level types. Furthermore, the type algebra allows different kinds of basic types composition and grouping to build complex types representation such as structures, unions, or types like "pointer to an array of 25 integer pointers" for example. Therefore, the implementation in the compiler is a rather straightforward translation.

On the code labels side, type generation is not that obvious. As code is meant to be executed in a dynamic context, label annotations need to describe a state, a context that must hold, i.e. be true, when execution reaches that point. To its largest extent, a context would hold the state of "everything" accessible by a running process, like processor registers, every byte of stack and heap memory, and all other possible resources. Obviously, describing "everything" is far out the compiler reach and doesn't make much sense in practice. So as the TAL approach suggests, we focus only on registers and stack state, which nevertheless hold all local variables in a C function and all values to be manipulated within functions. Heap usage tracking is part of our future research work.

Before annotating labels, code must be split into basic blocks. A basic block begins with a label, followed by zero or more assembly instructions, up to the next label or jump (conditional or not) whichever comes first. A jump to the following label is implied at the end of any block that doesn't explicitly end with a jump. The entire source code thus appears as a bunch of labelled blocks of code and the function concept becomes meaningless at this point. Also, code generated by LCC is not globally optimized and the x86 back end stores intermediate results and common subexpressions into temporary variables on the stack rather than registers. It can then be assumed that values are not carried across basic blocs boundaries by registers. The only exception is the EAX register that is often use to hold the return value of a function. Consequently, other registers are never considered as initialized at the execution point following any label (except for ESP and EBP which point to the stack). The registers annotation has thus been left out in code label annotations.

Typing the stack state requires a good knowledge of x86 back end stack management and a complete control flow analysis of the code. Stack management is similar to common commercial compilers. Control flow analysis computes the semantic content of this stack frame at the beginning of every basic bloc, i.e. at every label. A deeper analysis provides information about uninitialized, initialized or read-only variables, which is added to the semantic information to build the stack state type. In the case where multiple paths in the control flow graph lead to the same block and carry different information about variables state, we use a worst case approach. For example, a variable may or may not be initialized in different paths, therefore it is considered uninitialized where the paths meet in the control flow graph.

```
int function( parameters ) {
    variables declarations (A)
    complex code (uses temporaries)
    if ( condition )
    {
         variables declarations (B)
         simple code (no temporaries)
    }
    return value
}
```

Figure 1. Simple C code example.

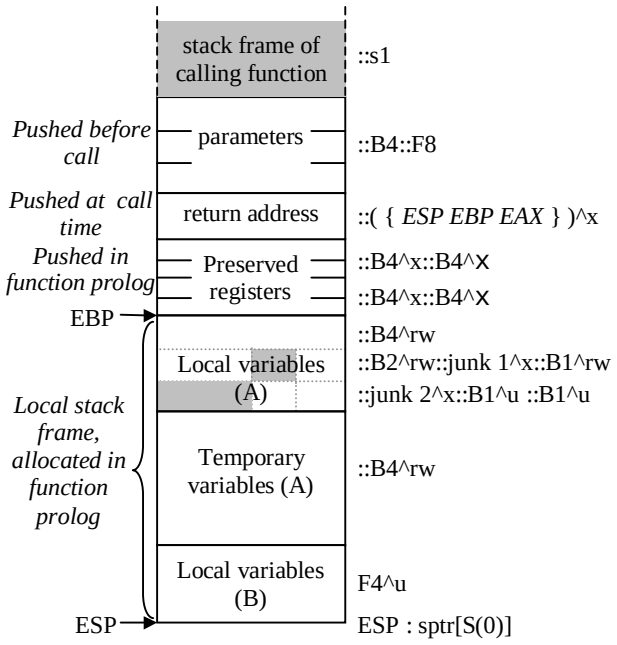

Figure 2. Stack configuration.

Figure 1 shows a simple C code example. Some variables are declared at the function level while some other are declared in an inner compound-statement. Also, some temporary variables are generated by the compiler to hold temporary results in the "complex code" part. Such variables are semantically equivalent to local variables. In this example, both EBP and ESP registers point to the stack frame

and therefore, are typed in label annotations. ESP always points at the end of the function stack frame or further if arguments are pushed onto the stack before a call, but EBP always points at the beginning of the stack frame, as shown in Figure 2.

This figure shows a hypothetical stack content when execution reaches the "simple code" point. The actual label annotation for this block of code describes among other things the type of the ESP register, which is shown on the right side of the figure, written from bottom to top. Roughly speaking, this type means that ESP points to the stack, where there is several variables of different types and access. Among them, the return address is expressed as the type of several registers that must hold on function return. Their actual type was removed from the figure given their complexity. It is worth noting however that ESP is also typed in this part of ESP type. A typed relation can then be expressed between the actual stack type and the caller stack type. The complete stack state of a running process is somehow the sum of local stack frames of all called and not yet completed functions since the process was started. As functions in the actually compiled source file may be called in an order that is well often not known statically, and since they might also be called from functions in other source files compiled separately, the complete stack state cannot be computed during compilation.

However, type variables are used in annotations as placeholders for yet unknown types. In the stack figure, the "s1" type variable holds for the stack content before the actual function was called, which can't be computed at this point. Actually, unknown types include the stack type before the actual function was called and an arguments list for functions such as `printf` which take a variable number of arguments. Other information from the calling function required to type both ESP and EBP registers is also represented by type variables. Namely, this other information is needed to type EBP and ESP as they should be at the beginning and the end of the function that is, before the local stack frame is allocated, and right after a return instruction is executed. Such information is only known within blocks that call the actual block, and is probably quite different from one block to the other if those blocks formerly belonged to different functions. Also, different labels may require a different kind of type variables. Each call or jump must therefore provide the appropriate information to complete the destination label type in a macro with parameters fashion. The verification process will take care of variables substitution.

4. Code Verification

Ultimately, the verification process will match a certificate against its corresponding binary executable to detect any malicious alterations to either one, and will then match this certificate against a local security policy to prevent unwanted behaviors from the executable. Any security policy includes de facto low-level security (safety) properties that deal mainly with memory and stack safety.

The actual implementation of both the compiler and the verifier only address the safety issue, as covering security properties is currently an ongoing work on both the compiler and the verifier. Furthermore, certified binary executable are not yet generated and we are still working on the assembly language output code along with its type file generated by the compiler. But since there is almost a direct relation between the assembly code and binary code, as well as between a type file and a certificate, our actual verifier does pretty much the same job as the targeted final verifier, although its implementation is quite simpler. Actually, safety properties covered by generated annotations and their verification are proper use of stack and CPU registers.

The verification process is split into three main steps: files parsing, type consistency check and instructions verification. Parsing was done using a Flex and Bison++ generated lexer and parser. Type consistency does some basic check about types in annotation that we assume to be of any given kind. For example, user-defined C types such as structures are stored in the type file as a "kind" and a "type". In that case, the kind specifies that the associated type is a "piece of memory of n bytes". The type consistency step computes the size of the structure in memory and matches it against the kind. This step prevents some malicious alterations in the type information and ensures that the compiler and the verifier are both correct or both wrong. Given the relative complexity of both process, it is unlikely that they are both wrong "in the same way". All parsed types must be consistent to continue with the next step.

The last step is the core of the verification process. Blocks of code are verified one by one, independently. Each block of code is a basic block, made of a label followed by zero or more assembly instructions. A label annotation describes a context under which all instructions are to be executed. Starting from this context, instructions are verified one by one, in order. A transition rule is associated to every assembly language mnemonic (instructions) known by the verifier. Some mnemonic might have different meanings depending on the type of their operand and are thus associated with more than one rule.

A transition rule states pre and post-conditions that must hold when executing the corresponding instruction. The preconditions describe what could be considered as valid operands, if any, and a context state that must be valid before executing the instruction. Post-conditions tell how the context is to be modified after executing the instruction.

Each instruction is verified by applying the corresponding transition rule down to the last instruction of the block,

which is always a jump, either explicit (a jump instruction) or implicit (in the case a jump to the following block of code is implied when the last instruction is not a jump). Transition rules for calls and jumps are slightly more complex than others since they must also substitute type variables in the destination label annotation and make sure that this new context (the modified version of the label annotation) can exist under the current context.

Any unverified condition automatically aborts verification with an error. No recovery scheme is actually implemented. Such an error may occur if there was malicious modifications of the assembly code and/or annotations, or if the code generated by the compiler is wrong. In fact, annotation are generated before the generated code, still in a structured form, is translated into assembly language by the back end. We could then detect some errors in the back end if there were any. But fortunately, the version of LCC we use seems so far to be error free.

5. Secure Self-Certified Code

As explained before, actual certifying compilers have focused on simple type safety properties. However, the security issue is of paramount importance. The consumer code accepting an untrusted code must check that the latter will not affect the secrecy (by leaking sensitive information), integrity (by corrupting information), authentication (by impersonating authorized principals), availability (by denying service to legal users), etc. Consequently, we need to extend our certifying compiler in order to consider the security aspects of a program.

As mentioned in [6, p. 530], the ideal system would contain both the compiler support and compact annotations of TAL, and the flexibility of PCC. On one hand, PCC uses a general-purpose logic, and can encode more expressive security properties. On the other hand, TAL provides compiler writers with a higher-level set of abstractions than PCC.

Our intended approach take advantages over both approaches (TAL and PCC).

First, on the code producer side, we use type annotations provided by TAL to generate a secure self-certified code. However, in order to consider security aspects of the code, we plan to extend the type system with effect discipline [9]. This extension consists of an inference type-system that propagates the security effects that result from accessing private resources, sending sensitive information over the network, etc. The idea is to consider security effects as part of the static evaluation of the code. The resulting annotations constitute valuable information for static analysis purposes, especially for verification, and correspond to the certificate that is delivered with the generated code.

Second, on the code consumer side, two verification process steps are needed. The first step corresponds obviously to the validation of the certificate. The purpose of this validation is to check, with certainty, that the certificate corresponds to the associated code. It is done by type-checking the code under the certificate constraints. The second step involves a model-checking verification technique. It consists of the extraction of a model from the information contained in the certificate. This model provides an abstraction of security aspects of the program. Afterwards, this model is matched against a logical specification of security policies. We plan to use a modal μ-calculus based logic to specify security policies.

Figure 3 shows the typical process of generating and verifying secure self-certified code.

6. Related Work

Recently, a surge of interest has been devoted to the security aspect of programming languages, especially through the compilation process. Moreover, we report two different research projects that are based respectively on TAL and PCC approaches.

In [10], the authors present an extension to the TAL approach that allows the generation of certified code for expressive security policies. In their proposed approach, security policies are specified using security automata [8]. One of the particularities of security automata is that its specifications can always be enforced by inserting run-time checks. Thus, during compilation, they dictate when to insert run-time checks. Especially, the compiler places run-time checks around every call to sensitive operations to ensure that the program uses these operations according to the specified security policies. The program is then optimized, removing run-time checks wherever it is possible to do so. The resulting code is then verified (type-checked) for its safety before being linked with a secure annotated interface of the system.

This approach seems to be very promising and useful. However, it is not clear how efficient the optimizer is in removing the dynamic checks. Furthermore, while security automata seem to be adequate to specify any *safety property* [8], it remains that they fail to capture some interesting security policies like information flow.

In [1], the authors propose an extension of the PCC approach to support high-level security policies. Among others, the main novelty in this research is the security policy component. They intend to bring together ideas from proof-carrying code with ideas from authentication logics. By doing so, they intend to build a system that is practical and handles sophisticated high-level security policies.

This approach has a number of advantages over other ones [1]. However, while the use of logics to specify se-

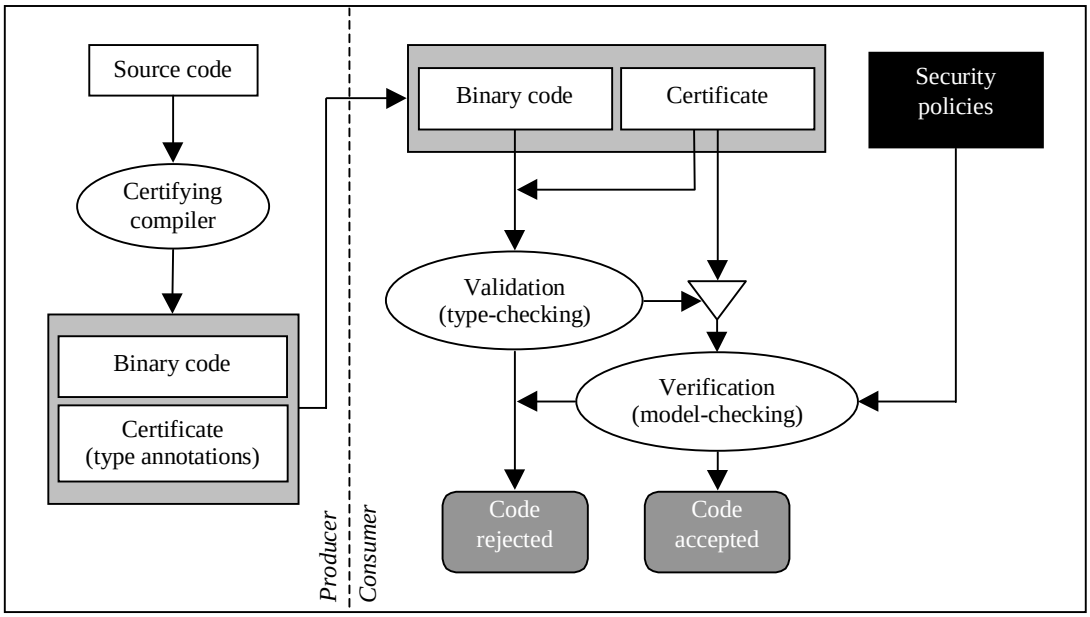

Figure 3. Secure self-certified Code.

curity policies seems to be more expressive than security automata, it is not clear how efficient authentication logics are to express security policies related to malicious code.

7. Conclusion

In this paper, we have reported our progress in defining and implementing a certifying compiler that produces a secure self-certified code that might be used to ensure both safety and security of the code. The current implementation of both the compiler and the verifier only addresses the safety issue, as covering security properties is currently an ongoing work. We intend to use a modal μ-calculus based logic to specify security policies and to address the verification problem through model-checking. The main idea is to match the model, extracted from the program through effect annotations, against a security policy specification.

As future research work, we plan to emphasize on the elaboration of the formal definition of the logic. Also, we aim at achieving efficient and practical program checking of potentially malicious code against security properties specifications. Finally, we hope to come up with practical tools and technologies that address the automatic detection of malicious code in COTS.

References

[1] A. W. Appel, E. W. Felten, and Z. Shao. Scaling proof-carrying code to production compilers and security policies. `http://www.cs.princeton.edu/sip/projects/pcc/whitepaper/`, Jan. 1999.

[2] J. Bergeron, M. Debbabi, J. Desharnais, B. Ktari, M. Salois, and N. Tawbi. Detection of malicious code in COTS software: A short survey. In *First International Software Assurance Certification Conference (ISACC'99)*, Washington D.C, Mar. 1999.

[3] C. Fraser and D. Hanson. *A retargetable C Compiler: design and implementation*. Adison-Wesley, 1995.

[4] D. Kozen. Efficient code certification. Technical Report TR98-1661, Cornell University, Jan. 1998.

[5] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, Atlanta, GA, USA, May 1999.

[6] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[7] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997.

[8] F. B. Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, Jan. 1995.

[9] J. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[10] D. Walker. A type system for expressive security policies. Technical Report TR99-1740, Cornell University, Apr. 1999.