SEYED HEIDAR PIRZADEH TABARI

Encoding the program correctness proofs as programs in PCC technology

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en informatique
pour l'obtention du grade de Maître ès sciences (M.Sc.)

Faculté des sciences et de génie
UNIVERSITÉ LAVAL
QUÉBEC

2008

Résumé

L'une des difficultés de l'application pratique du code incorporant une preuve (*Proof-Carrying Code* (PCC)) est la difficulté de communiquer les preuves car celles-ci sont généralement d'une taille importante.
Les approches proposées pour atténuer ce problème engendrent de nouveaux problèmes, notamment celui de l'élargissement de la base de confiance (*Trusted Computing Base*): un bogue peut alors provoquer l'acceptation d'un programme malicieux.
Au lieu de transmettre une preuve avec le code, nous proposons de transmettre un générateur de preuve dans un cadre générique et étendu de PCC (EPCC) (*Extended Proof-Carrying Code*). Un générateur est un programme dont la seule fonction est de produire une preuve; c'est-à-dire la preuve que le code qu'elle accompagne est correct. Le générateur a pour but principal d'être une représentation plus compacte de la preuve qu'il sert à générer. Le cadre de EPCC permet l'exécution du générateur de preuve du côté client d'une manière sécurisée.

Abstract

One of the key issues with the practical applicability of Proof-Carrying Code (PCC) and its related methods is the difficulty in communicating the proofs which are inherently large.
The approaches proposed to alleviate this, suffer with drawbacks of their own especially the enlargement of the Trusted Computing Base, in which any bug may cause an unsafe program to be accepted.
We propose to transmit, instead, a proof generator for the program in question in a generic extended PCC framework (EPCC). A proof generator aims to be a more compact representation of its resulting proof. The EPCC framework enables the execution of the proof generator at the consumer side in a secure manner.

Acknowledgments

I am greatly indebted to my advisor, Danny Dubé, for his many contributions to this work; he is a thoughtful person and a real problem solver. He has allowed me much freedom to explore many different ideas throughout my masters studies and has encouraged all of my aspirations. Danny's guidance and insights has been an invaluable asset to my research, and I sincerely hope that this work reflects his usual high standard for quality.

I would also like to thank the other members of my M.Sc. committee: Béchir Ktari and late Clermont Dupuis for their encouragement and for spending their precious time to read my thesis and provide comments, advice, and questions.

Josée Desharnais, Béchir Ktari, Mohamed Mejri, and Jean-Marie Beaulieu must be thanked as professors for their understanding and their excellent classes at Laval University. I interacted with Béchir more than the others. There were many times I have visited his office for various problems. He was always reachable, his door was always open. Though we did not use the NWCC compiler, I would like to thank Nils Weller for helping me to learn how NWCC works.

I would also like to thank all the people in the administrative offices at Laval University -especially Lynda Goullet for helping me out with all the paperwork.

My deepest thanks to my parents without whom I would not be here. They have provided me with loving support throughout my childhood and adulthood, and have been a great source of inspiration. I am very grateful to my sister Setareh for her constant love and support.

Last and most significantly, I express my love and appreciation for my wife, Sara Shanian, who tirelessly supported me and encouraged me throughout my masters studies. Her contributions in my life are endless.

To my parents, my sister, and my beloved wife.

Contents

List of Figures

Chapter 1

Introduction

1.1 Software Security

The rapid growth of the Internet and networks goes along with a rising need for security for users. Instant access to a huge number of untrusted and malicious softwares through Internet on one hand and lack of security due to hardness, expense to set up, and annoyance to get along with, on the other hand, makes it easier for intruders to implement their plans.

The latest Internet Security Outlook Report issued by CA Inc. details the extent of the malicious softwares problem [3]. It argues that malicious softwares (malwares) are becoming more sophisticated, and increasing use of obfuscation techniques to hide in plain sight, will help criminals conceal their activities. Just in year 2007 the malware volumes grew by 16 times. It also warns that, since much of the criminal activity is targeted at nations where there are large populations of Internet users, malware and software failure problem is an international issue.

Furthermore, the requirement of fully trusted software for ultra-expensive and vital projects clearly shows the lack of required technology basis to address these new needs of computer security [1]. Since the methods used to implement security policies are less expensive and more flexible in software than in hardware, security is increasingly becoming a software issue [4].

Recently, there have been some explorations on the domain of programming languages to find techniques that can help us enforce the necessary security policies to computing systems. We refer to these explorations as language-based security ap-

proaches [5]. In other words, language-based security as defined by Schneider is "a set of techniques based on programming language theory and implementation, including semantics, types, optimization, and verification, brought to bear on the security question." [5]

By that definition, the approach proposed by this thesis falls within the framework of language-based security.

1.2 Security design principles

Language-based security conceptually is a combination of two classic computer security principles:

1.2.1 Least privilege

About thirty years ago, Jerry Saltzer and Mike Schroeder described a design principle, known as Principle of Least Privilege [7]. According to this principle, throughout execution, each principal should be given the minimum resources necessary to accomplish its task. In other words, depending on the context, every process, user or program must be able to access only such information and resources that are necessary to its legitimate purpose.

As a result of this restricted access, the untrusted code will not have the means and rights to perform a system-level operation in order to use vulnerability in other applications to exploit the whole system. Indeed, least privilege results in a system with better security. Moreover, the restriction on sensitive system operations in this principle leads to better system stability.

As a real-world example of the principle of least privilege we can refer to "need to know" restriction rule in military. Under need-to-know restrictions, even if one has all the necessary official approvals (such as a security clearance) to access certain information, one would not be given access to such information unless one has a specific need to know; that is, access to the information must be necessary for the conduct of one's official duties (e.g. Battle of Normandy in 1944) [8].

1.2.2 Minimum trusted computing base

The trusted computing base (TCB) of a computer system is the set of components in which the occurrence of bugs might put the security properties of the entire system in danger [9]. Rationally, in a system, the smaller the TCB is, the less probable the compromise in security would be. Thus, the best way to assure that a system is secure is to keep its TCB small and simple.

If an attacker can replace or modify any components of the TCB or, simply, there exists a buggy component in TCB, the TCB can no longer be trusted. Conceptually, the Minimum trusted computing base principle can be applied to any trust management system, where the trusted party is the TCB and the trusting party is the computing system.

An important property of the principle of Least Privilege and Minimal TCB is their independency of the architecture, computer speed and other variable dimensions of computer systems. This makes them remain sound and sensible throughout the time.

1.3 Current approaches

Our daily life is becoming more and more dependent on computers. We have access to networks and to a huge number of untrusted softwares through Internet. Suppose we wish to download and run a program from an unknown or untrusted source. One of our biggest concerns would be about the protection of our computers from this untrusted code. The following classes of approaches are currently used to reach this goal.

1.3.1 Authentication

The idea in Authentication is that one accepts only the codes that come equipped with a signature of a trusted producer. Ideally, since this producer is trusted, the code can be also trusted. Microsoft Authenticode is a popular example of a code-signing mechanism to assure users that software they download from the Internet is authentic and has not been tampered with [10]. This class of approaches has several drawbacks.

Applying this approach in large scale, as a solution to the untrusted codes problem, is against the second principle of security design. As it was mentioned earlier, the principle

of Minimum trusted computing base can be applied to any trust management system, where the trusted party is the TCB and the trusting party is the computing system. Hence, considering the trusted companies as the TCB, in order to decide whether a code can be trusted or not, we need to add each and every trusted company to the TCB which leads to enlargement of the TCB. Trusting a producer does not necessarily mean that the produced code is safe, and performs as it claims. As a popular example, we can refer to Microsoft Word 97 which contained a hidden pinball game [11]. Thus, it is clear that we need approaches which can ensure that the untrusted code can cause no harm to the computing system.

1.3.2 Code Analysis

One way to ensure that the untrusted code can cause no harm to the computing system is to understand the code behavior. *Code analysis*, also known as *program analysis*, is the process of focusing in on specific aspects of a code to gain an understanding of the code behavior. This process of automatically analyzing can be done in different period of the code lifetime (compile time, execution time).

Dynamic Analysis

Dynamic code analysis techniques observe the execution of a code and perform an appropriate action before the code violates the security policy. The performed action against the security violation can be discontinuation of the execution of the code, auditing the code, or simply making a log file. The SASI [12], Naccio [13], and Polymer [14] are among the best-known approaches in dynamic analysis.

Execution monitor

The systems that perform the dynamic code analysis are called *execution monitors*. In dynamic analysis, it is important to create a safe analysis environment also known as a *sandbox*. Possible analysis environments are the operating system (Figure 1.1(b)), the code space (Figure 1.1(c)), and a wrapper program (Figure 1.1(d)).

The operating systems may have distinct user rights management, timer interrupts, and access control list for protecting operating system resources (e.g., files or devices) from unauthorized access by humans or malicious code. An access control list is the

usual means by which access to services is controlled. It is simply a list of the services available, each with a list of the entities (users, programs, processes) permitted to use the service. Currently, most desktop machines are configured as single-user so applications have complete access to the machine resources.

When the code space is the analysis environment, some lines of code should be inserted before each memory access and each control transfer to ensure that those accesses are valid.

Wrappers, interpreters, and virtual machines are environments that intercept (and interprets or redirects) the instructions issued by the wrapped code. The wrapped code does not execute directly on the underlying hardware but instead is interpreted by another program. Every wrapped code instruction is thus executed only after it has been checked and found not to be violating the security policy being enforced. In this way, a broad range of security policies can be applied.

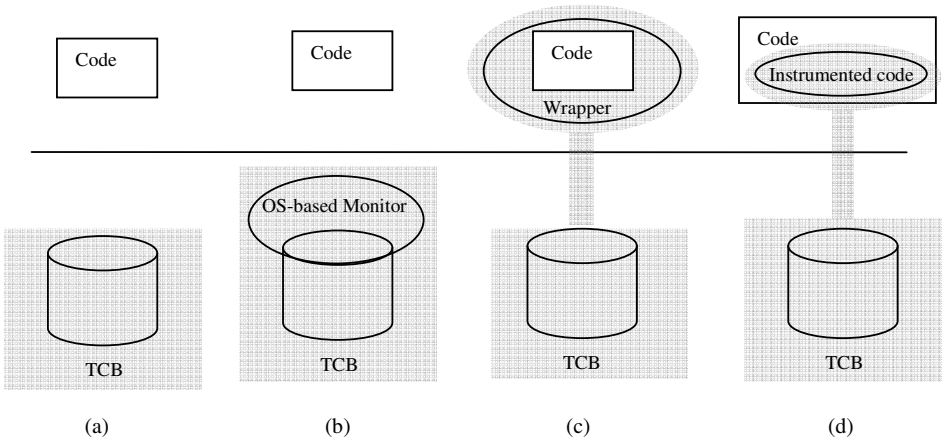

Figure 1.1: The TCB of execution monitors

Execution monitors (EMs) are usually easy to implement and they can work with binary codes which makes them language independent. Despite these strong points, the EMs suffer form some drawbacks. The system resources are engaged by the monitor the whole time the code is running and even if one run of the code was safe, we can not be sure about the next runs. Thus, the monitor should work every time we run the code for the whole running time which results in a big overhead. Another disadvantage of the monitor lies in its usual fail-stop behavior. Fail-stop treatment does not fit well in projects where the price of stopping the program is high (e.g., satellite navigation system [1]). Furthermore, it is of high importance for an approach to be in accordance with the principles of security design. Obviously, execution monitors are in partial accordance with the least privilege principle as they are intended to perform such task. However, assuming the hardware and the base computing system as the TCB (Figure 1.1

(a)), unfortunately, the growth in the size of the TCB is inevitable for all three types of execution monitors regardless of their implementation.

Static Analysis

As we mentioned in the previous section, one of the major drawbacks of the execution monitors lies in their fail-stop behavior against the malicious codes. The fail-stop behavior, itself, is because of the runtime investigation which leaves the monitor with no way but to stop the code in order to avoid any security violation to happen. The stoppage then may lead to a great loss in time, money, or even other forms of security in systems where the renewal of the untrusted code is hard or even impossible (safety-critical or huge scientific projects in outer-space may end up with denial of service due to the stoppage of a piece of code as a result of a simple mistake in the code [2]).

Furthermore, the execution monitor can only confirm the safety of the current trace of the code execution up to the current moment in the trace. While, for a code to be safe, the set of all possible traces of the code should be proven safe. Therefore, through execution monitoring the safety of the code can not be proven.

Thus, it would be reasonable to do the code analysis without actually executing the code (as opposed to dynamic analysis). A family of techniques of code analysis in which the code is analyzed by tools to produce useful information without actually executing programs built from that code is called Static analysis.

Static analysis techniques find out a code's possible behavior prior to its execution, rejecting a code whose set of possible behaviors includes unacceptable behavior. This a priori understanding of the code is the same as proving the code safety. This can be done by providing the computing system in destination (i.e., the code consumer) with a safety prover which can verify the code safety upon receiving the code. Figure 1.2 shows the TCB for the general static framework in which the growth in size of TCB depends on the size of Safety checking system.

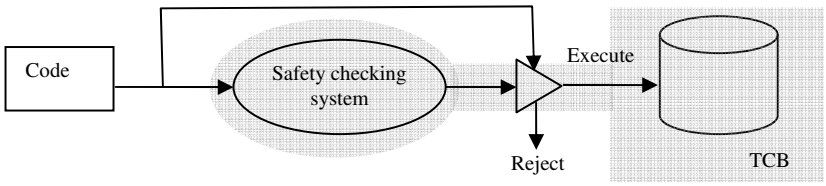

Figure 1.2: The TCB in general static analysis framework

Typed assembly language (TAL) [15, 16], Flint [35], and Proof-carrying code (PCC) [17], are among the best-known approaches in static analysis.

Proof-Carrying Code

A safety verification system should be capable of verifying the safety of every untrusted code. Since verifying the safety of the received code is a hard task, it inevitably would need a complex verifier. "The complexity is the worst enemy of security", that is, placing a big and potentially buggy program in the TCB compromises the security of the computing system.

In order to ease this situation, Necula and Lee introduced the Proof-Carrying Code (PCC) approach [20, 17, 18, 21]. The main idea behind this approach is to shift the roots of the problem out of the TCB. This is done by breaking the verifier system into two components: a complex Safety Prover and a simple Proof checker, placing the Safety Prover on the producer side and the Proof checker on the consumer side (i.e., in TCB).

In this way, the burden of proving the safety of the untrusted code is shifted on the shoulders of the producer. Therefore, in this framework, the producer first proves the safety of the code then attaches the safety proof to the code and sends it to the consumer. On the consumer side and before executing the code, roughly, the proof checker checks the code against the accompanying proof. Upon success, the code could be executed.

As it is shown in Figure 1.3, PCC has a relatively small TCB. Moreover, since PCC does the static analysis, once the safety of an untrusted code is checked successfully before the first run, there is no need to check the code before the next runs. As a result, we will have a computing system with less overhead and more security [22, 23]. These benefits nominate the PCC as one of the strongest frameworks to be used in mobile-code security. However, unfortunately, this great approach suffers from some shortcomings. Apart from the difficulty of building or generating the proofs for the code, one of the crucial obstacles for the practical applicability of the Proof-Carrying Code technique is the size of the proofs that must accompany the code. To be precise, the difficulty of communicating the proofs which are inherently large makes the PCC less scalable.

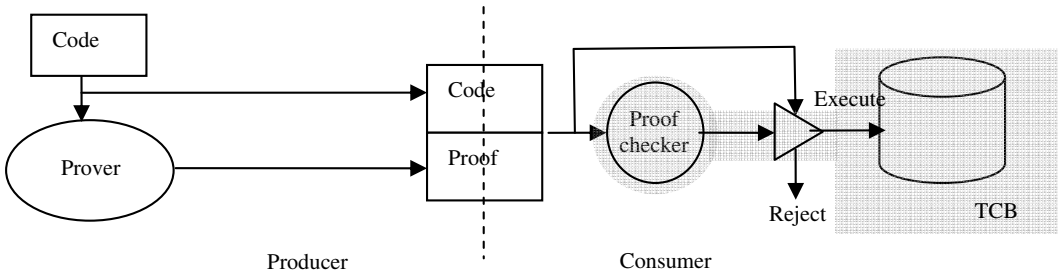

Figure 1.3: The TCB in simplified PCC framework

1.4 Thesis contribution

As discussed above, the presence of untrusted and malicious codes and the absence of the necessary security bases and frameworks for safe use of the mobile codes are among the main concerns of our computer-dependent world. The problem with the current approaches is either because of their refusal to obey the security principles or due to their hard applicability.

The trustworthiness of the proof-carrying code is an important advantage over approaches that involve the use of complex security systems on the consumer side. However, this approach has scaling issues because the size of the safety proofs increases quickly. The approaches proposed to alleviate this suffer with drawbacks of their own especially the enlargement of the Trusted Computing Base, in which any bug may cause an unsafe program to be accepted.

In contrast to these developments, we have worked on a hybrid approach with some modifications in the framework level which essentially solves the PCC's scalability issue. In our approach, instead of transmitting the proofs, a proof generator for the code in question is sent. The new modified extended framework enables the execution of the proof generator on the consumer side in a secure manner. To cut a long story short, this thesis makes the following contributions to the field of mobile code security and verification.

- We show the design of a generic extended framework for proof-carrying code (EPCC). We show that the designed framework is tamperproof and can resolve some key issues of the previous frameworks.
- We present the design of a safe and small virtual machine (VEP) introduced in the EPCC framework. For this, we implemented the VEP to work as an online reference monitor in EPCC framework.

- We show empirically that the EPCC and its conjoint virtual machine (the VEP) can be used in an industrial-strength framework. To show this, we implemented the necessary programs to complete the end-to-end chain. For this, we implemented an assembler for the VEP byte code. We also implemented a C compiler to target the assembler language. Finally, in order to have a runnable EPCC framework, we made use of GUNzip to build a sample proof generator.

1.5 Thesis outline

The remainder of the thesis is organized as follows. In Chapter 2, we present our generic Extended Proof-Carrying Code (EPCC) framework and take a glance over related works and assess each approach merits and drawbacks. Chapter 3 discusses the design of the *V*irtual machine for *E*xtended *P*CC (VEP). It describes the trade-offs we had to make when designing the VEP and discusses the way in which the VEP works. In Chapter 4, we make the whole system work by bridging from theory to practice. In this chapter, we present a sample implementation of an EPCC framework. Finally, we summarize and give an outlook of future work in Chapter 5.

Chapter 2

Extended Proof-Carrying Code

In this chapter, we present our generic Extended Proof-Carrying Code framework. First we introduce and motivate Proof Carrying Code (PCC) along with mentioning its limitations. Then, we take a glance over related works and assess each approach's merits and drawbacks from our perspective. Finally, we propose our new approach.

2.1 PCC-based Approaches

In PCC-based approaches, an untrusted code producer must convince a code consumer of the safety of his code by sending a proof. In this section, we peruse the Proof-Carrying Code and its limitations together with its similar approaches. It is worth mentioning that we visualize the approaches using data flow diagrams. Rectangles denote components taking data represented in ovals as input or output such data. Arrows indicate how the data flows and the policies are shown by vertical scrolls.

2.1.1 PCC: Characteristics, and Obstacles

Characteristics

Proof-Carrying Code [20, 17, 18, 19, 21] refers to a static analysis approach in which the consumer has the ability to verify a proof of general safety properties of a code before executing the code.

Since the verification is done before the execution of the code, like other static analysis approaches, PCC enforces the security conservatively. This conservative behavior is due to the lack of the access to useful runtime information (like inputs and variable values). Thus, using PCC can lead to some false negatives. Figure 2.1 shows how PCC takes care of received codes. The darker region shows the codes which are accepted as safe codes by PCC while the white section between the origin axis and the darker region represents the safe codes which are not accepted by PCC, mentioned as false negatives.

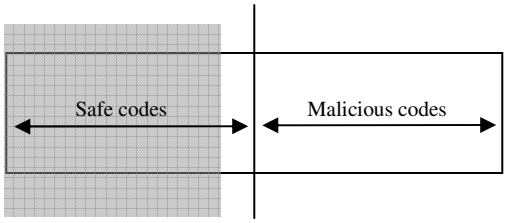

Figure 2.1: Safe codes in PCC point of view

The false negatives is the price that PCC pays to gain an assured security. This clearly shows the importance of the certainty of security as one of the raisons d'être of the PCC approach.

To find out other important aspects of PCC approach, we briefly talk about the original framework of the PCC.

In a PCC system, there are typically two main parties, (1) a code producer, who builds machine code along with its safety proof, and (2) a code consumer, who wishes to run the compiled code as long as it satisfies the safety policy. In reality the communication between these two parties is more complicated and consists of a multi-step interaction between the producer and the consumer. In the first step, the producer sends a program consisting of the code and additional annotations. These annotations consist of loop invariants and function pre- and post-conditions and provide more information about the code. This additional information makes the following step of the procedure easier.

In the next step, the consumer applies the Verification Condition Generator (VCGen) to the received annotated code, according to his/her particular safety policy to generate a verification condition. A verification condition is a logical formula that, if satisfied, implies that the code satisfies the safety policy [18]. Here, additional annotations can be used to make the Verification Condition Generator's job easier. Generating a verification condition is a straight-forward procedure which makes the VCGen a simple and fast program. The consumer then sends the generated verification condition to

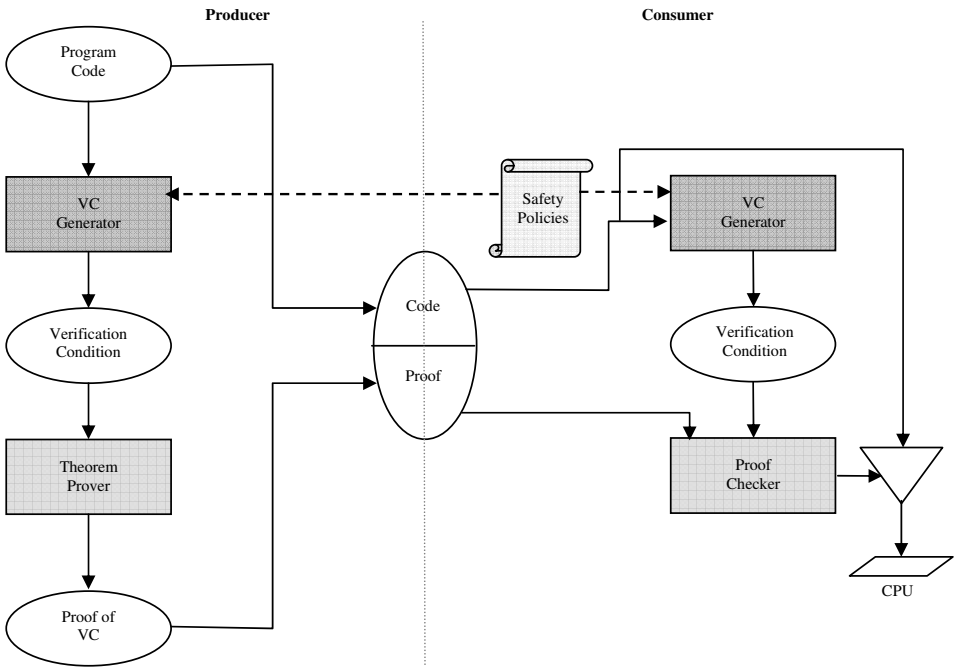

Figure 2.2: Traditional Proof-Carrying Code framework

the producer.

The producer runs a theorem prover (in many cases along with necessary human interventions) to get a proof of received verification condition. The theorem prover uses the axioms defined as part of the consumer's safety policy. Thus, a proof of the verification condition constitutes a proof of safety of the code with respect to the consumer's safety policy.

In general, proving the verification condition is resource-consuming which can result in low performance. Furthermore, considering that the theorem prover is a complex program, it could not be placed on consumer side. Therefore, in PCC the theorem prover is on the producer side. Thus, it is the beauty of this technique that the difficult and complex job is not being done by the code consumer during any steps of PCC.

In the next step, the producer submits the proof to the consumer. The code consumer checks the proof before executing the code submitted by the producer. Therefore, he should verify if the received proof is really a proof of the verification condition. The code consumer runs the proof checker to verify that the proof is indeed a valid proof of the verification condition constructed in the previous phase. If the proof check succeeds the consumer can then execute the code safely. Once the safety of an untrusted code is checked successfully, there is no need to check the code before the next runs.

As it is shown in Figure 2.2, it is possible to simplify the dialogue between the code producer and the code consumer, assuming that they share a same VCGen (the code producer has a copy of the VCGen). In this way, the code consumer receives the annotated code attached to its proof and runs the verification condition generator. Then, he checks the proof against the verification condition and if the check succeeds he can execute the code safely.

One of the most important properties of the PCC framework is that the PCC programs are tamper-proof. That is, an intruder cannot modify the code or the proof in a way that results in execution of a malicious code on the consumer side. Any attempts to tamper with either the code or the proof results in a validation error at proof checking. In the few cases when the code or the proof are modified such that the validation still succeeds, the new code is also safe.

TCB Components

As we mentioned earlier, PCC intends to have a small trusted computing base by shifting the hard task of proving the safety of the code to the producer side. Figure 2.3 shows the main components of the TCB in original PCC framework. In practice, the trusted computing base in the PCC framework is composed of the followings.

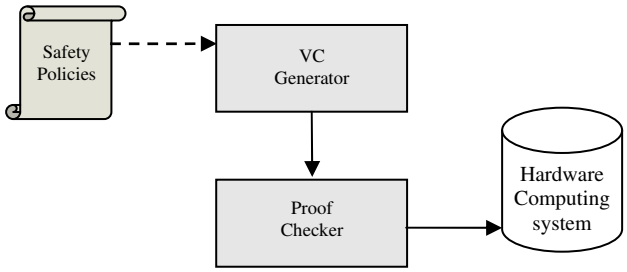

Figure 2.3: Trusted computing base in PCC

- Hardware: we suppose that the actual machine hardware will operate as expected.
- Logical structures:
  - Logic: The logic is used to express the safety policies and proof in a formal way. Proof of the soundness of the logic is done by hand, so a small and simple logic would be more convenient.
  - Safety Policy: The policies which are expressed in the syntax of the logic and should be respected by the untrusted code. The safety policy itself is combined of two main parts:

  - Proof rules: The rules which are necessary for proving the verification condition. Two sample proof rules are represented in Figure 2.4(a).
  - Decoder: The decoder interprets the semantics of an individual instruction and results in the local safety condition for executing that instruction and a set of possible machine states resulting from the execution of the instruction.

- Proof checker: An implementation of the PCC logic which makes it possible to express statements in the syntax of the logic and mechanically check proofs, is called Proof-checker. The proof-checkers are usually simple programs and the proof checking is a straight-forward task. Assuming logic has all the desirable properties, we must trust that the proof checker implementation is correct.

- Verification Condition generator: VCGen is responsible for handling the control-flow aspects of the code and will result in a verification condition whose validity entails the safety of the code. As it is shown in Figure 2.4(c), the VCGen computes the verification condition of the entire program by combining all of the local safe-progress conditions (safety conditions of each instruction) identified by the decoder. For this, the weakest pre-condition of the program, starting form the post-condition and work back, is computed. Figure 2.4(d) shows the pre-condition and the post-condition of the sample source code presented in Figure 2.1(b). Figure 2.4(e) shows a sample definition of the VCGen (we refer interested readers to [19] for detailed descriptions of the definitions).

Obstacles

The total size of the mentioned components constituting the TCB in proof-carrying code approximately is about 15000 to 20000 lines of code. Any bug in these components can compromise the security of the whole system. It is often possible to use the elusive standard of "residual defect density" as a metric for faultiness to measure the number of faults that remain in a software code at the delivery point. A typical target in software development is to achieve a residual defect density of less than one error per one thousand lines of non-comment source code (KLOC) [37, 38]. However, leading edge software development organizations typically achieve a defect density of about 2 defects/KLOC [39].

Even if the TCB has a residual defect density between 1 and 2, the TCB's number of lines of code is relatively large and can not easily be trusted. That is, the anxiety

$$\frac{m \vdash e_1 : \texttt{int} \qquad m \vdash e_2 : \texttt{int}}{m \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{}{m \vdash 0 : \texttt{int}}$$

(a) Sample proof rules

```
fun sum (l : T list) =
 let
  fun foldr f nil a = a
    | foldr f (h::t) a = foldr f t (f(a, h))
 in
  foldr (fn (acc, Int i) => acc + i
          | (acc, Pair (i, j)) => acc + i + j)
        l 0
 end
```

(b) Sample source code

$$VC(\Pi, Inv, Post) = \forall \mathbf{r}_i. \bigwedge_{i \in Inv} Inv_i \supset VC_{i+1}$$

(c) The verification condition for the entire program

$$\begin{array}{rcl} Pre & \equiv & \mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{T list} \\ Post & \equiv & \mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{int} \end{array}$$

(d) The pre-condition and the post-condition if the sample code

$$VC_i = \begin{cases} [\mathbf{r}_s + op/\mathbf{r}_d]\, VC_{i+1}, & \textit{if } \Pi_i = \text{ADD } \mathbf{r}_s, op, \mathbf{r}_d \\ \mathbf{r_m} \vdash \mathbf{r}_s + n : \texttt{addr} \wedge [\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s + n)/\mathbf{r}_d]\, VC_{i+1}, & \textit{if } \Pi_i = \text{LD } \mathbf{r}_d, n(\mathbf{r}_s) \\ (\mathbf{r}_s = 0 \supset VC_{i+n+1}) \wedge (\mathbf{r}_s \neq 0 \supset VC_{i+1}), & \textit{if } \Pi_i = \text{BEQ } \mathbf{r}_s, n \\ Post, & \textit{if } \Pi_i = \text{RET} \\ \mathcal{I}, & \textit{if } \Pi_i = \text{INV } \mathcal{I} \end{cases}$$

(e) Sample verification condition generator definition

* The figures are taken from the "Proof Carrying Code" by G. Necula (POPL'97)

Figure 2.4: Verification condition generation

about the TCB grows along with its number of lines. Therefore, to have a safe and implementable PCC framework, one of the obstacles in front is its relatively large TCB.

Besides the problem of the large TCB, there is also the matter of proof size. In principle, the proofs can be exponentially large, as mentioned by Peter Lee: "...as a general matter, the size of the binaries is an issue that must be addressed carefully" [23]. Thus, one of the crucial obstacles for the practical applicability of Proof-Carrying Code and related techniques is the size of the proofs that must accompany the code. It is important to have a compact representation of proofs because they are possibly sent through communication networks. In traditional PCC framework, it was not unusual to see proofs that were 1000 times larger than the associated code which made the use of PCC impractical for all but the tiniest examples [27].

Another issue in PCC framework is that it does not provide the producer with

enough flexibility. That is, the producer is constrained to submit a proof in a logic which has been imposed by the consumer. That is, even if the producer find it possible to build a simpler proof in a higher-order logic, he is forced to build the proof in the consumer's logic which might result in an overweight proof.

To sum up, the PCC technology has the following obstacles in front:

1. proofs are large (practicality);
2. TCB is relatively large (security);
3. producer should do the hard work with inadequate means (flexibility).

Any solution to combat these obstacles and to make a refinement in the PCC technology should respect the following fundamental characteristics of the PCC approach:

1. give the highest priority to the security (raison d'être);
2. intend to have a small TCB;
3. leave the easier tasks to the consumer;
4. can not be tampered with.

In the following sub-sections we summarize two other PCC-based approaches proposed to combat the mentioned obstacles. We try to make the different approaches comparable by highlighting two aspects: the trusted computing base size and the size of the safety proofs in those systems.

2.1.2 Foundational PCC

In order to relieve the second drawback discussed in the foregoing sub-section, Appel introduced the notion of foundational proof-carrying code (FPCC) [28].

Although the TCB components in traditional PCC framework are simple, Appel points out that the VCGen (and consequently the TCB size) is too large [28] and it needs to be verified itself.

To explain more clearly, if there exists a bug in either the VCGen or the typing rules then the TCB becomes vulnerable. As a matter of fact, a proof-carrying code certifying compiler for Java named Special-J [24], happened to have errors in its typing rules, discovered by League *et al.* [30]. Needless to say, this bug affects the overall safety of the PCC system.

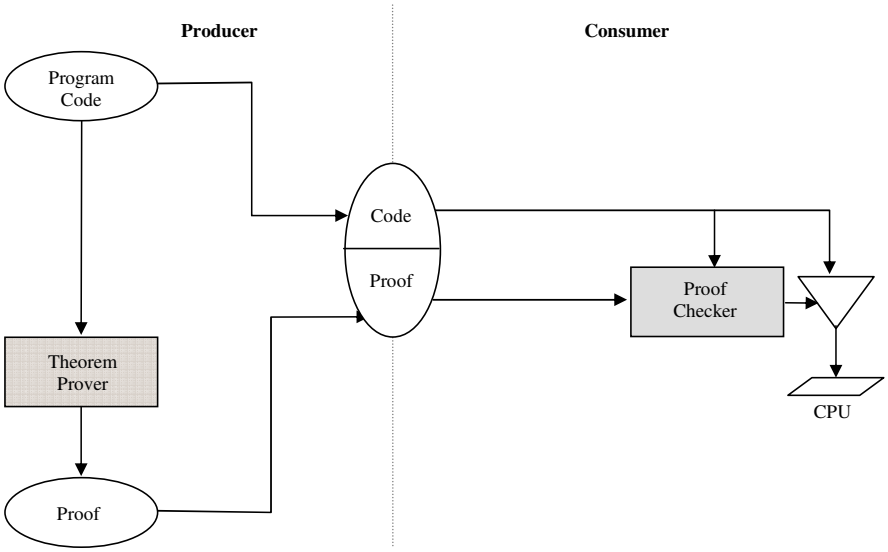

Figure 2.5: Foundational proof-carrying code framework

Foundational Proof-Carrying Code aims to further reduce the TCB size by an order of magnitude. That is, emphasizing a minimal TCB, they removed the VCGen and the safety policy from the consumer side, as it is shown on Figure 2.5

Foundational PCC is based on the idea of defining the semantics of the machine instructions and the proof rules using only a foundational mathematical logic. In this way, Appel *et al.* avoid using the VCGen by defining the operational semantics of machine instructions and the safety policies in a higher-order logic. This is done by modeling the machine instruction with a transition from one machine state (set of memory and registers) to another machine state. Then, the safety policy is defined in the following way: "A given state is safe if, for any state reachable by an arbitrary sequence of legal instructions, there is a safe successor state." Hence, a code is safe if we get a safe state. For this, "*type checking rules are proved as lemmas. Then, the proofs are constructed and checked once per system, and rules are used many times to check programs. In this way, a safety proof is the application (derivation) of type checking rules and it shares proofs of common lemmas*" [29]. FPCC uses a higher order logic with few axioms of arithmetic, from which it is possible to build up most of modern mathematics.

There are three main components in a Foundational PCC system: a theorem prover, a proof checker, and a safety proof of the code. The theorem prover should produce a proof of safety to be accompanied by the code. The proof checker verifies the safety proof before the program gets executed.

Safety Proof Size

The proofs in Foundational PCC, in comparison with traditional PCC, are more complicated to produce and as Appel himself stated, can explode exponentially. Therefore, the proof size which is a crucial obstacle for the practical applicability of Proof-Carrying Code and related techniques is remained unsolved. According to Necula, the proofs size in Foundational PCC is 20% bigger than the proofs size in traditional PCC. This makes the proof communication harder and the use of Foundational PCC even less practical than the traditional PCC [27].

Trusted Computing Base

Foundational PCC is concerned with minimizing the trusted computing base of the system, including not the VCGen as shown in Figure 2.6. FPCC, in principle, is strictly more secure than traditional PCC because it has a smaller trusted computing base. With this technique, Verification Condition Generator is removed from the TCB, and the TCB becomes minimal.

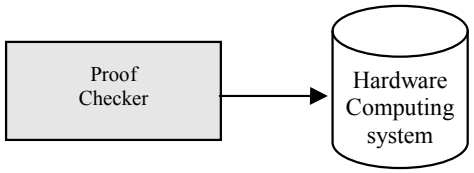

Figure 2.6: Trusted computing base in FPCC

2.1.3 Oracle-Based PCC

One of the main impediments to scalability in traditional PCC is that the proofs can be very large. In order to alleviate this problem, Necula proposed a new strategy called Oracle-based Proof-Carrying Code (OPCC) [27]. In his new approach, the way in which the proofs can assist the verification on the consumer side is changed. As it is shown

in Figure 2.7, this change in strategy, led to a change in the framework, namely, they assumed that the consumer uses a non-deterministic proof checker.

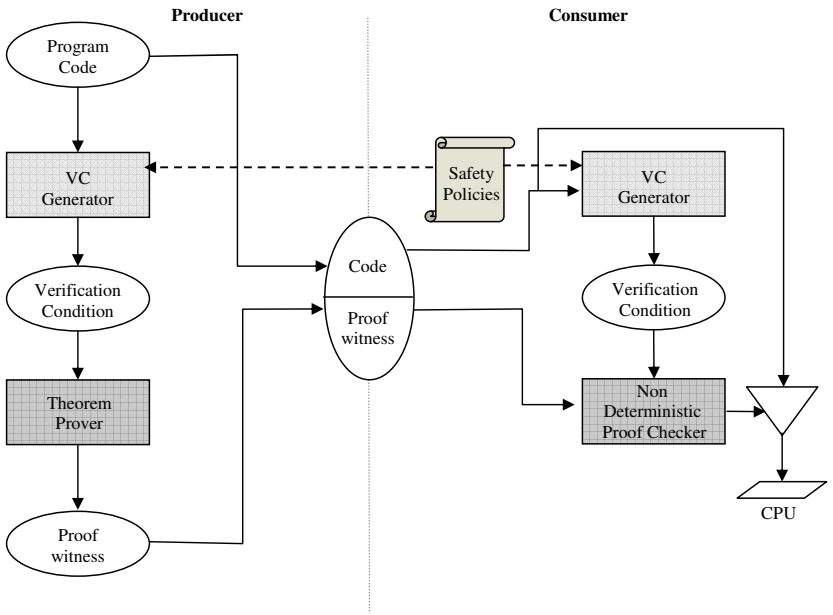

Figure 2.7: Oracle-based proof-carrying code framework

In order to make use of the new non-deterministic proof checker, they replaced the proof by an oracle string which guides the non-deterministic checker. Every time the checker must make a choice between the possible ways to proceed, it consults some bits from the oracle.

To be more precise, the untrusted theorem prover on the left-hand side records a sequence of bits that shows which sub-goals failed and needed backtracking. Then, the producer sends this bit-stream to the consumer. On the consumer side, the received bit-stream works as an "oracle" which can be used by the trusted non-deterministic proof checker to avoid backtracking. It goes without saying that the oracle, like proofs in PCC, needs not be trusted. That is, if the oracle is wrong, then the trusted checker will go wrong, and will fail to find the proof.

In this approach, the trusted non-deterministic proof checker, in fact, is a non-deterministic theorem prover. This theorem prover is given the task of proving the verification condition. Whenever the prover has to pick from n choices, it reads some bits from an oracle string to resolve that choice. As a result, the oracle is used to drive the theorem prover to a final proof without search, and as such, the oracle string can be considered as a proof witness.

Safety Proof Size

The oracle-based proof-carrying code is efficient. Experimental evidence shows that oracle strings, as suggested by Necula, can be about 1/8 of the code size and about 30 times smaller than proofs in traditional PCC [27]. However, Wu [31] found the code size relation deceptive: "Unfortunately, this statistic is somewhat misleading. [...] a machine language program and a proof witness. The SpecialJ proof-carrying Java system on which Necula measured oracle-based checking transmits three components: The machine code, the proof, and a Java "class file". The Java class file, as is usual in any Java system, contains descriptions of the types of all procedures (methods) in the program (untrusted code), including formal parameter and result types. However, the "1/8 size" figure does not include the Java class files".

While the small size and low cost of checking an oracle string are appealing, a potential problem with them is that there are no currently known ways to manipulate or compose them. Thus, oracle strings for subprograms might be hard to use directly when trying to find certificates for larger programs (oracle strings are based on guiding the search for cut-free proofs). They are also fragile in the sense that small changes in the formula to be proved or in the version of the theorem prover can invalidate an oracle string.

Trusted Computing Base

The downside of Oracle-based PCC is that, as it is shown in Figure 2.8, it involves complex trusted components, such as a type system with axiomatic rules for memory safety and the VCGen and the non-deterministic proof checker. Any flaw in the implementation of these components can compromise safety of the system.

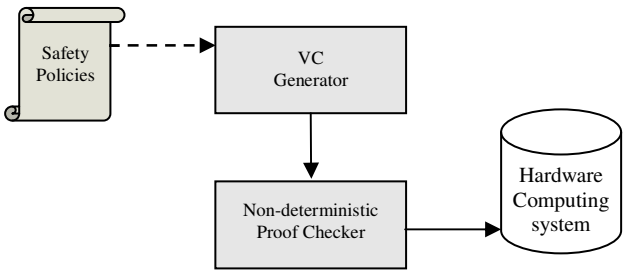

Figure 2.8: Trusted computing base in OPCC

The trusted computing base in Oracle-based PCC is about 26000 lines of code which is bigger than the TCB size in traditional PCC. As mentioned in the PCC obstacles,

and as the second principle of security design suggests, any bug in the TCB may cause an unsafe program to be accepted. For example the Special-J system, showed a critical leak in its type axioms [30]. Unfortunately, one can find the big size of the TCB in OPCC against the first and the third characteristics of a PCC approach, as mentioned in Section 2.1.1.

2.2 Extended Proof-Carrying Code framework

In this section we study the Extended Proof-Carrying Code framework. First, we explain the idea behind our proposed approach. Then, we present the framework and talk about its properties.

2.2.1 Sending a Proof generator

As we mentioned earlier, one of the crucial issues for the practical applicability of Proof-Carrying Code and its related techniques is the size of the proofs that must accompany the code. Therefore, it is desirable that proofs be represented in a compact format. One way to reach this goal is *Proof optimization* in which the proofs are built in a more compact form and can be interpreted as proof of the original form [32, 33]. The best proof optimization approaches result in proofs which are 15-30 times smaller than the original proof and pay the price of the enlargement of the TCB [27, 31, 26]. We are not in favor of compromising the security of the system by a big TCB expansion simply because the proofs are too large.

Another way of compacting the proofs is through *Data compression*. Data compression techniques try to find more compact representations for data, from which the original data can be reconstructed exactly. Many such algorithms compress data by searching for more efficient encodings that take advantage of repetition in the data. These techniques are not well exploited in PCC framework due to the following reasons. The consumer of compressed data must first decompress it, this needs a safe decompresser on consumer side. Generating the proof of safety for a normal decompressor (relatively big program with about 3000 lines of code) is a difficult task not worth performing because such decompressor would be a specific decompresser that can not have the potential to work with a proof compressed by an appropriate but different compressor. That is, to gain the advantage of a good compression, each time, the safety of a new decompresser should be proven according to the compression method

which is appropriate for the safety proof of a code.

We present in this thesis an extended framework that allows the PCC proofs to be represented as programs. This helps us not to pay a proof-size price and enables the PCC to handle even very large programs. The idea behind the new framework, which we are going to present, is inspired by the Kolmogorov complexity. We introduce the notion of Kolmogorov complexity in the following sub-section.

2.2.2 Kolmogorov complexity

Roughly speaking, the Kolmogorov complexity of a string is the shortest computer program that produces the same string, i.e., that computes it, prints it, and then halts. One important observation is that this measure of complexity indicates how much a string (or, in the context of proof-carrying code, a proof) can be compressed: the ideal compressed form for a given proof is the shortest program that outputs that proof.

Formally, the Kolmogorov complexity $K_{\mathsf{U}}(x)$ of a string x is defined as the length ℓ of the shortest program capable of producing x on a universal computer U such as a Turing machine. This complexity is incomputable.

$$K_{\mathsf{U}}(x) = \min_{p \in \{0,1\}^*} \{\ell(p) \colon p \text{ on } \mathsf{U} \text{ outputs } x\}$$

The definition depends on the specific computer programming language and the universal computer that is used. We define these two components according to our generic extended PCC framework which we present next.

2.2.3 Extended Proof-Carrying Code framework

The idea behind the Extended Proof-Carrying Code (EPCC) is simply to send the proof in the form of a program. In this way, we make it possible for the producer to send a proof generator instead of the proof, where according to Kolmogorov complexity, the proof generator ideally can be the shortest program which can output the original proof. For this to work, the consumer should be capable of running the proof generator on a universal computer, in a secure manner, and obtain the proof.

Proposed generic framework

In order to benefit from the above idea in an organized manner, we proposed a generic EPCC framework. A diagram of an EPCC system is given in Figure 2.9. In an EPCC system, there are two main parties, a code producer, who sends a code along with its safety proof generator, on the left-hand side and (2) a code consumer, who wishes to run the code, provided that it is proven safe by the system on the right-hand side.

The communication between these two parties may consist of a multi-step interaction between the producer and the consumer depending on the proof-carrying code framework that they extend. Generally, at the first step, the producer runs a theorem prover to get a safety proof of the code he intends to send. Here, in contrast with other PCC frameworks, the consumer is not forced to generate the safety proof in the logic that the consumer imposes.

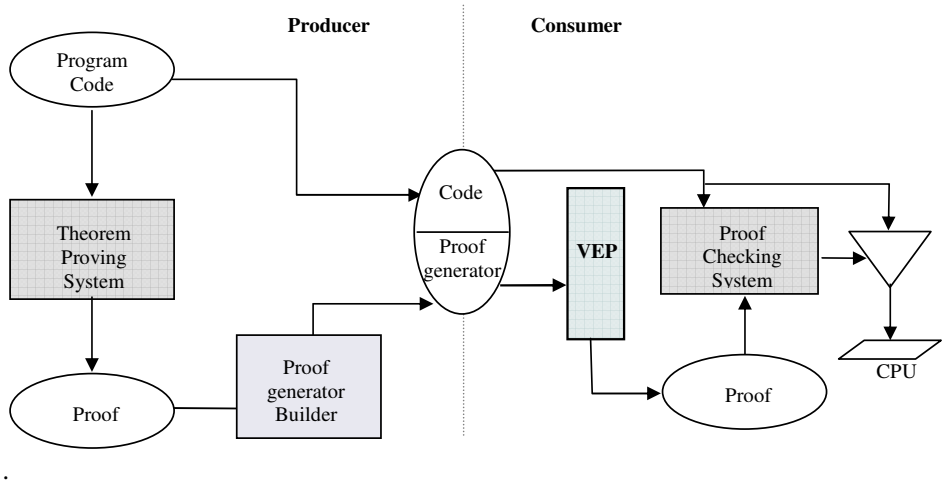

Figure 2.9: The framework of the generic Extended Proof-carrying code

The producer can use this opportunity to build the proof in a logic (e.g., a higher-order logic) that results in a smaller proof. In other words, the producer has the possibility of reducing the size of the safety proof by using a custom logic which can be later converted (translated) to the logic set by the consumer.

Then, the producer writes a proof generator. In accordance with the Kolmogorov complexity, this proof generator can, in principle, be the shortest program which can output the safety proof in the format which is acceptable to the consumer. That is to say, the generic EPCC framework provides the producer with the opportunity of compacting the proof in two steps of optimization and compression.

In the next step, the producer submits the code accompanied by its safety proof generator to the consumer. The consumer is required to check the proof before executing the code submitted by the producer. Therefore, he runs the safety proof generator on the Virtual machine of EPCC (VEP) and obtains the safety proof. Then he runs the proof checker. After the proof check succeeds the consumer can repeatedly execute the code safely. As one can easily observe the EPCC framework like the PCC is tamper proof.

One of the crucial components in the EPCC framework is the VEP which is a part of the trusted computing base of the EPCC. Safe execution of the proof generator depends on the safety of the VEP and the way it imposes the security requirements. Here, we advert some important aspects about the VEP, and later, in Chapter 3 we study the design of the VEP thoroughly. In the following, we discuss the ways in which the VEP provides us with the necessary basis for applying Kolmogorov complexity idea and enables the execution of the proof generator at the consumer side in a secure manner.

The VEP: A Universal Computer

A *universal computer* is a computer which is capable of universal computation. That is, given a description of any other computer or program and some data, a universal computer can perfectly emulate this second computer or program [34]. The best-known contender for the title of universal computer is the *Turing machine*. A Turing machine is a computing machine which has a number n of one-way infinite *tapes*, divided into *cells*, one next to the other. The cells of the tapes can be blank or contain a symbol from some finite alphabet. The first of the tapes is known as the input tape, on which a string of symbols is written, and the last of the tapes is known as the output tape where the result of the Turing machine for that input is written. The other $n-2$ tapes can be thought of as auxiliary tapes. On each tape the Turing machine has what is called a *head*. At any one time, a head sits on a particular cell and can read the symbol which is written on that cell, write a symbol onto that cell and move to the left or to the right or stay put (in some models the tape moves and the head is stationary). It is worth mentioning that a Turing machine can equivalently process a single infinite tape.

The current implementation of the VEP is a stack-based machine which is equivalent in computing power to a Turing machine. The VEP reads the code and performs actions on its Stack and Heap. Here, the code space can be regarded as a read-only tape and the stack as a modifiable tape (since the VEP provides us with random access to the stack cells). The Heap is to provide the programmers with an extra flexibility.

Knowing that the VEP has finite resources, pops up the question if it can be considered a universal computer destination for the proof generator according to Kolmogorov complexity. The answer is yes, it is possible because in a finite amount of time, a universal computer can only manipulate a finite amount of data which fits in finite resources. In this way, the VEP can be considered as a universal computer destination for the proof generator.

The VEP: An Execution Monitor

The proof generators in EPCC framework are untrusted programs which have to be executed on the consumer side. Since running untrusted programs on consumer side is against the raison d'etre of the PCC approach and can compromise the security of the system, we need a security mechanism for running the proof generator safely. For this to happen, the VEP should provide a tightly-controlled set of resources for proof generators to run in. Network access, the ability to inspect the host system, or reading from input devices and writing into file streams should be disallowed. In this sense, the VEP ought to be an execution monitor.

As we mentioned in Section 1.3.2, two main drawbacks of the execution monitors are their high overhead and their fail-stop manner of encountering unsafe codes. Here, we discuss the existence of each of these issues.

As for the overhead, in execution monitoring, the system resources are engaged by the monitor the whole time the code is running and even if one run of the code was safe, we can not be sure about the next runs. Thus, the overhead is the result of the system resources engagement by the monitor for each and every period of the code execution. Overheads are usually quantifiable "costs" of some kind. If we denote the cost as C and the total cost of a monitor as C_T, we can show C_T the total cost of an execution monitor EM as:

$$C_T(EM) \approx \sum_{n=1}^{m}(t_{EM}\ avg(C(EM)))$$

where n is number of the times the execution monitor EM runs, t_{EM} shows the execution time period of the EM and m is the total number of runs and $avg(C(EM))$ is the average cost of running the monitor EM per CPU cycle which can be defined as:

$$avg(C(EM)) = \frac{\sum_{t=1}^{t'} C_{EM}^{t}}{t'}$$

where C_{EM}^{t} is the cost of running the execution monitor EM at any CPU cycle t for

the period t'. Now we can formulate the problem as follows:

$$\lim_{m \to \infty} (C_T(EM)) = \infty \tag{2.1}$$

$$\lim_{t_{EM} \to \infty} (C_T(EM)) = \infty \tag{2.2}$$

As it is shown in Equations 2.1 and 2.2, an unbounded number of runs and execution time of the execution monitor each can place an unbounded cost on the system which uses the execution monitor.

In the case of EPCC, we need the execution monitor VEP to run only for a single time, in which the proof generator outputs the proof or fails. Therefore, for the Equation 2.1, number of runs m is bounded to 1. Now, if we can run the monitor for a limited period of time we can bound the Equation 2.2. For that reason, the VEP runs for a limited number of CPU cycle, which is set in a beforehand agreement between the producer and the consumer, and checked during the execution of the code. Hence, the problem can be bounded as follows:

$$C_T(EM) \leq (t_{EM} \ \max(C(EM))$$

In this way, the VEP can enforce fine-grained memory safety, control-flow safety, and type safety through execution monitoring with an insignificant constant cost.

As for the second drawback of the execution monitors, the fail-stop manner is aligned with the safety of the EPCC framework. That is, we need the VEP to act in a fail-stop manner to prevent an unsafe proof generator to continue its execution. Therefore, not only the fail-stop manner has no dangerous consequent, but also it is required. Thus, the major drawbacks of using the execution monitors are negligible when using the VEP as an execution monitor.

The VEP: Accordance with Security Design Principles

It is of high importance for an approach to be in accordance with the principles of security design. Obviously, the VEP and other execution monitors are in partial accordance with the least privilege principle as they are intended to perform such task.

In addition to this natural accordance of the VEP with the least privilege principle, we designed and built the VEP in a way that it assigns only such resources that are necessary for the proof generator to perform its legitimate purpose. This is done through

an agreement in which the producer and the consumer settle the possible amount of resources that can be used by the proof generator. Among these resources are the heap, the stack, and the code space of the proof generator. Any disobedience of the agreement by the proof generator is doomed to discontinuation of its execution. In this way, the VEP puts the principle of least privilege strictly into practice.

With regard to the second principle of the security design, we set a criterion for the size of the TCB. The criterion was to design and build the VEP in a way that the enlargement of the TCB be less than the difference between the size of the TCB in Oracle-based PCC and the size of the TCB in traditional PCC in terms of the lines of code. That is, we aimed to implement the VEP such that the security of EPCC be stronger than Oracle-based PCC according to the second principle of security design.

The size difference between the two versions of the TCB in traditional PCC and Oracle-based PCC is about 2000-3000 lines of code. Interestingly, the current version of the VEP is less than 300 lines of code which is much smaller than the standard we set. Since the VEP consists of small number of lines it can be verified easily by pen and paper. Furthermore, in principle, the VEP does not need to increase the size of the TCB if it would be possible (without difficulty) to prove it safe in a PCC framework.

2.3 EPCC Applications

In this section, we present some of the possible applications of the EPCC framework. For this, we start by studying the benefits of employing EPCC on traditional PCC framework (i.e., extending the traditional PCC framework in a way that it can accept a proof generator). Then we propose employment of EPCC for FPCC and OPCC as two other PCC techniques and their possible benefits. It is important to mention that only the first application which is an EPCC version of the traditional PCC framework is implemented as a part of this work (detailed information about the implemented framework on Chapter 4) and the two other EPCC frameworks are presented as feasible propositions.

Extending traditional PCC

In Figure 2.10, the consumer side in traditional PCC is shown on the left-hand side. In PCC, the proofs are an order of magnitude bigger than the code size. The EPCC

version of traditional PCC is shown on the right-hand side. The dialogue between the producer and the consumer remains the same as traditional PCC except for some minor modifications. In extended version of traditional PCC, instead of accompanying the code with a safety proof, the producer accompanies it with the a safety proof generator which he has built and custom-made earlier. On the consumer side and upon reception of the proof generator, the consumer safely executes the proof generator on the VEP and obtains the proof. The generated proof is then given to the proof checker. The proof checker checks the generated proof against the verification condition and if the checking is successful, the consumer can run the code safely.

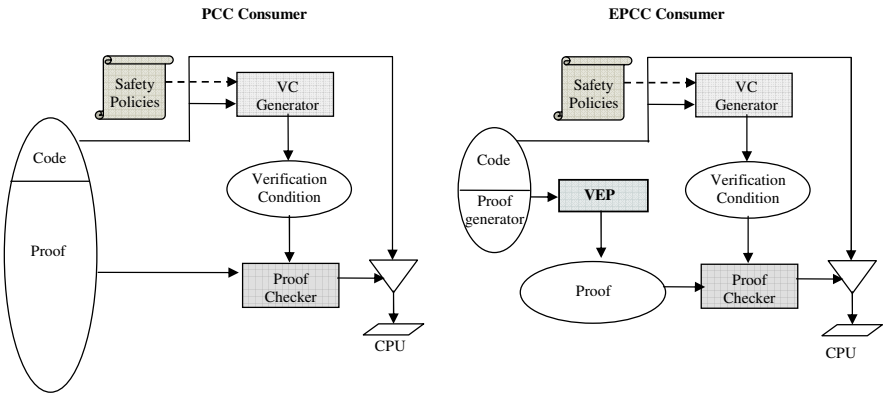

Figure 2.10: Consumer side in PCC versus its extended version

The safety proofs in PCC are represented in Edinburgh Logical Framework (LF) [36]. A logical framework is a formal system in which other logics can be readily represented. The typical LF representation of the proofs are large, due to a significant amount of redundancy. Storing these proofs in a format that requires less space than usual (e.g., compressing them) would alleviate the problem of proof size in communications, because it enables devices to transmit or store the same amount of data in fewer bits. Lossless data compression techniques work best on data with repetition in its representation. Therefore, the fact that proofs contain many repeated patterns of proof rules and redundant arguments, makes them suitable for lossless data compression. To gain a better compression, the data compression algorithm can be custom-made in keeping with the content of the proof which is going to be sent.

In our experiment by using the new strategy of EPCC, with an off-the-shelf compression technique, the type safety proof generators average 5% the original proofs which is about 30 times smaller than before. Interested reader can check the Chapter 4 to obtain more information about the results and the end-to-end implementation of the EPCC.

By extending the original PCC framework, we provide the producer with the pos-

sibility of sending a proof generator. This gives a chance to the producer to build a compact and specialized proof generator which can output the same proof on the consumer side. In this way, the proof size issue can be alleviated while the parties are provided with a more flexible framework in which the original logic of the proof generator can be different than that of the generated proof.

Extending Foundational PCC

Figure 2.11 shows the consumer side in the Foundational PCC framework on the left-hand side and its EPCC version on the right-hand side.

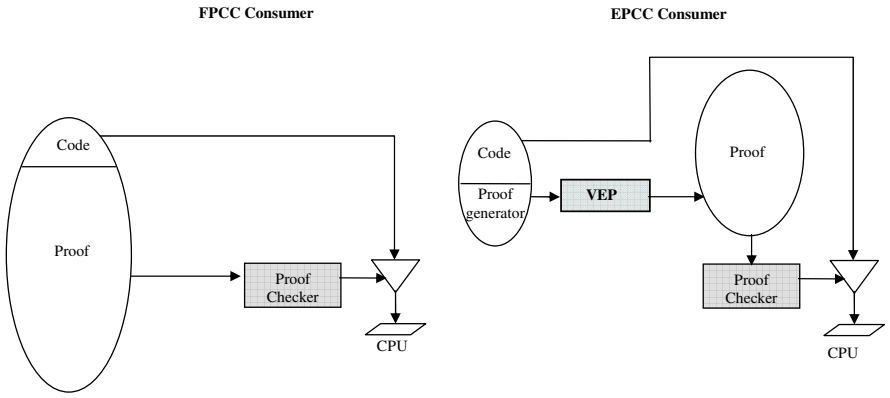

Figure 2.11: Consumer side in FPCC versus its extended version

In FPCC framework, the proofs are bigger than the proofs in traditional PCC which makes the scalability of FPCC harder. By extending the FPCC framework, the producer can send a proof generator whose size can be a fraction of the original proof size. In this way the crucial obstacles for the practical applicability of FPCC can be alleviated. Since the size of the VEP in Extended FPCC is smaller than the size of the VCGen traditional PCC framework, in principle, extended FPCC could be immediately more secure than the traditional PCC because it has a smaller TCB. The dialogue in extended FPCC is similar to the one in FPCC, except that in extended FPCC the consumer executes the proof generator on the VEP to obtain the proof.

Extending Oracle-based PCC

In Figure 2.12, the consumer side in Oracle-based PCC is shown on the left-hand side. To have a better understanding of the Trusted computing base size issue, the TCB is shown by a bigger square. The extended version of the Oracle-based PCC is shown on

the right-hand side. On this side, the code is accompanied by a proof generator. The consumer can execute the proof generator on the VEP and obtain the proof. Then the proof checker checks the generated proof against the verification condition and if the checking was successful, the consumer can run the code safely.

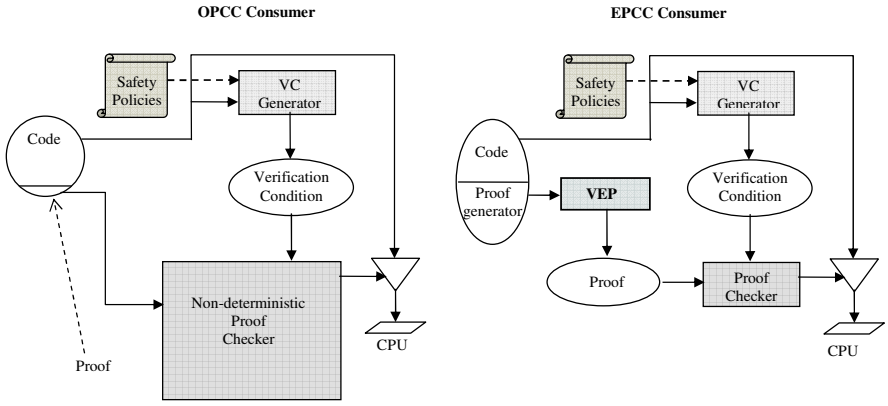

Figure 2.12: Consumer side in OPCC versus its extended version

By extending the Oracle-based PCC framework, we provide the producer with the possibility of sending a proof generator. The proof generator can use the oracle idea to generate the complete proof as the output. Since the VEP is smaller than the size difference between non-deterministic proof checker and the original PCC proof checker, the TCB size issue can be alleviated while the parties are provided with a more flexible framework in which the original logic of the proof generator can be different than the generated proof.

2.4 Overview

The Extended Proof-Carrying Code framework is to make the PCC idea more scalable and practical by alleviating the proof size issue while respecting the characteristics of the PCC technique.

EPCC provides the code consumer with the luxury of using a safe environment in which a big class of proof generators can be executed in a secure manner, regardless of the original logic in which the proofs were represented. In this way, EPCC leaves the easier tasks to the consumer and gives adequate means to the producer to do the hard task. This major flexibility for the consumer and producer, in addition to the alleviation of the proof size issue, are gained through a minor TCB extension of less than 300 lines of code which can be verified easily by pen and paper. In this way, EPCC intends to have a small TCB and give the highest priority to the security.

Chapter 3

The VEP Virtual Machine

This chapter discusses the design of the *V*irtual machine for *E*xtended *P*CC (VEP). It describes the trade-offs we had to make when designing the VEP and discusses the way in which the VEP works.

3.1 Machine Design

In this section, we present the design process of the VEP. A virtual machine is a functional simulation of a computer and its associated devices [41], which is implemented by adding a software to an execution platform to give it the appearance of a different platform which may have an instruction set that differs from that implemented on the underlying real hardware. Figure 3.1 shows the idea of using virtual machine by the

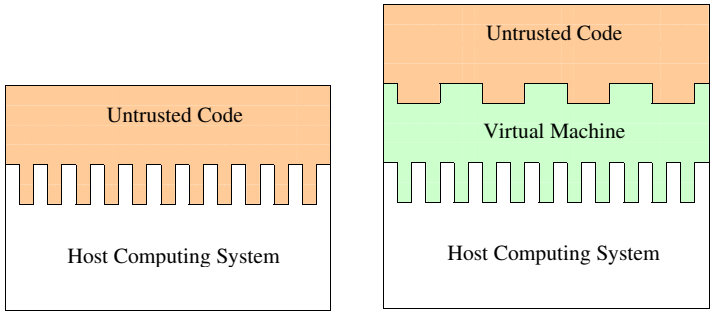

Figure 3.1: Virtual Machine

lego-like diagrams. Without a virtual machine, the untrusted-code gets interlocked with the host computing system through the *instruction set architecture* shown as intercon-

nections. On the other hand, a Virtual machine which is the virtualizing software, can translate (completely or partially) the instruction set architecture of the original platform, so that the untrusted-code sees a different instruction set architecture from the one supported by platform. That is, a virtual machine can work as a (partial or complete) emulator which executes programs written for the virtual machine instruction set on a machine that executes a different instruction set. Having restricted instruction set (e.g., the unnecessary instruction which gives the potential to write unsafe codes are omitted) and safe emulation (i.e., performing necessary checks before executing an instruction) by the virtual machine both can improve the security of the system. In this way, a virtual machine can be used to increase security, provide enhanced performance and simplify software migration.

3.1.1 Captured goals and requirements

The virtual machine design process starts by capturing the requirements. In EPCC framework, we execute the proof generator on the VEP. The proof generator can be a package of a decompression algorithm and the compressed proof. In this way, by executing the proof generator, the consumer is actually decompressing the compressed proof. We used the GUNZip algorithm as a representative of algorithms within the decompression techniques area. As a guideline, we tried to design the VEP in a way that it can support an efficient execution of programs written in a broad range of language.

The requirements of a virtual machine are mainly concerned with the properties such as: size, portability, performance, memory consumption, scalability, security, etc. In the case of the VEP, we dealt with the following requirements:

1. The VEP should provide us with a platform which has the potential of working with the Kolmogorov ideal compressor. According to the Kolmogorov complexity, this ideal compressor runs on a universal computer.

2. It should enable the execution of the proof generator at the consumer side in a secure manner. That is, the VEP should provide a tightly controlled set of resources for proof generator. Network access, the ability to inspect the host system, or read from input devices and write into file streams should be disallowed. Therefore, the VEP should be able to perform execution monitoring.

3. As indicated in EPCC framework, the VEP is a part of the TCB. Knowing that any bug in TCB can compromise the security of the whole system, we need the

VEP to have a size and simplicity, such that, it be feasible for a human to inspect and verify it by pen and paper.

4. The proof generators are sent in the language of the VEP. Consequently, the language of the VEP is a factor which can affect the size of proof generator. It would be helpful if the VEP can provide us with small size of code.

5. In EPCC framework the producer sends a proof generator to the consumer side. The consumer should run the proof generator on his side to obtain the proof. Considering that any execution has an overhead, code execution performance is a concern.

6. A virtual machine can die a quick death because no one writes codes specific to that VM. This could be due to its high complexity. Therefore, we want the complexity of the VEP to be low, and design it in a way that has the potential to become popular.

7. Since changing the VEP instruction set and memories in the future might improve its efficiency and performance, it would be nice to just leave room for future extensions to help the scalability of the VEP.

The mentioned goals and requirements are not equally important to us. The three first items of the above list are of very high importance. There also exist trade-offs between the list items. For instance, the low complexity and small code size, both depend on the number of instructions in the VEP instruction set; having small set of instructions results in a virtual machine with low complexity and, on the other hand, a big list of instructions makes the code smaller. Although these two factors are contradictory, there can be a good balance between them. As a result, finding a good trade-off is our major endeavor. For this, in making a design trade-off, we favor the more important over the less important cases.

Now that the established set of functional requirements is presented, we can discuss the design choices in the following subsections.

3.1.2 Machine Type

This subsection is related to the VEP machine type design choices: virtual register machine or virtual stack machine. Conventionally, a VM can either be stack-based or register-based. In a stack-based machine most operands reside on the stack, and operation commands pop their operands from the top of the stack and push their results

back onto the top of the stack. In a register-based machine most operands reside in (virtual) registers and the number of registers is limited.

In order to decide between these two design choices, we will keep an eye on the captured requirements. Implementing a universal computer can be done with a stack machine which has more than one stack or has one stack with random access. Nevertheless, register machines can be universal computers, therefore, both approaches can satisfy the first requirement.

Simplicity is another factor which is important to us. Although many virtual machines such as Parrot VM [46], VM of Lua 5.0 [44], and Rain VM [45] have been implemented with register instruction set architectures, the most popular virtual machines, like Java Virtual Machine [47] and Common Language Runtime [48], use a stack machine type rather than the register oriented architectures used in real processors, due to the simplicity of their implementation. Hence, an stack-based VEP helps us to better fulfill the third requirement in the list.

Furthermore, the simple stack operations can be used to implement the evaluation of any arithmetic or logical expression, and any program written in any programming language can be translated into an equivalent stack machine program. Moreover, the stack machines are easier to compile to. Potentially, this could prevent a quick death of the VEP to happen as stated on the sixth requirement of the list.

The last among the reasons which led us to choose the stack machine type over the register one is the properties of the compiled code in these two types of machine. A compiled code for a stack machine has more density than the one for the virtual machine. Davis et al. [52] translated Java Virtual Machine stack code to a corresponding register machine code. The resulting register code after elimination of unnecessary instructions in register format was around 45% larger than VM stack code needed to perform the same computation. This can specially affect the size of the proof generator written for the VEP. Accordingly we chose the stack machine type over the register one.

3.1.3 Instruction Set Architecture

The *Instruction Set Architecture* (ISA) of a virtual machine is the VM interface to the programmer. In the case of the VEP, available data types, the set of memory spaces are defined by ISA. The ISA definition also includes a specification of the set of opcodes (machine language) and the VEP's instruction set. Next, we discuss each of these parts and their design choices.

Data Types

The sequence of bits in the memory can represent many types of data: addresses, numbers, characters, and logical values True, False. Each instruction requires the correct type of data to produce a meaningful result. In the case of the VEP, we have two distinct types: *numbers* and *pairs*. Considering that we have machine word size data (e.g. 32 bits), as it is shown in Figure 3.2 the rightmost bit of the cell shows the data type of the stored value in that cell. This bit is not visible to the programmer while the remaining 31 bits are visible. From now on we use the term *cell* for the mentioned visible bits, and *type bit* for the mentioned invisible bit.

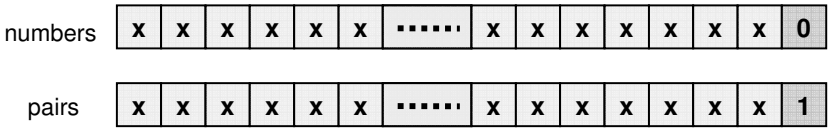

Figure 3.2: Data types in the VEP

If we have a cell that references a *pair*, the content of the cell represents the address of a pair in the heap memory. For a cell with its type *number*, the content of the cell is a signed integers. As it is the most common method of representing signed integers on computers, the numbers are represented in *two's complement* format.

$$-x := NEG(x) + 1$$

That is, the negative of a binary number represented by switching all ones to zeros and all zeros to ones and then adding one to the result. It is worth mentioning that the lowest byte of a number is at the rightmost side of the cell. Here, assuming that the machine word is 32 bits, the range of the number data type is between -1073741824 to 1073741823. By convention, the VEP interprets zero as False and all other values as True.

Memory

A virtual machine may use the memory of the platform in different ways. Here, we discuss how the VEP uses this memory. As mentioned in 3.1.2, for the VEP to be able to perform universal computation, it can have multiple stacks or a stack with random access. Current version of the VEP has one stack with random access. We also provide the programmers with additional flexibility by supplying the VEP with some additional memory as heap. Moreover, the VEP has a code space for the received code. Therefore, our machine uses three blocks of memory: a code space (made of byte-length cells), a

heap (made of machine word length cells), and a stack (made of machine word length cells). The stack in the VEP is ascending (i.e., the top of the stack is at highest address) and conforms to the full stack convention (i.e., the stack pointer points at the top of the stack) and first item on the stack is stored at bottom of the stack at address zero.

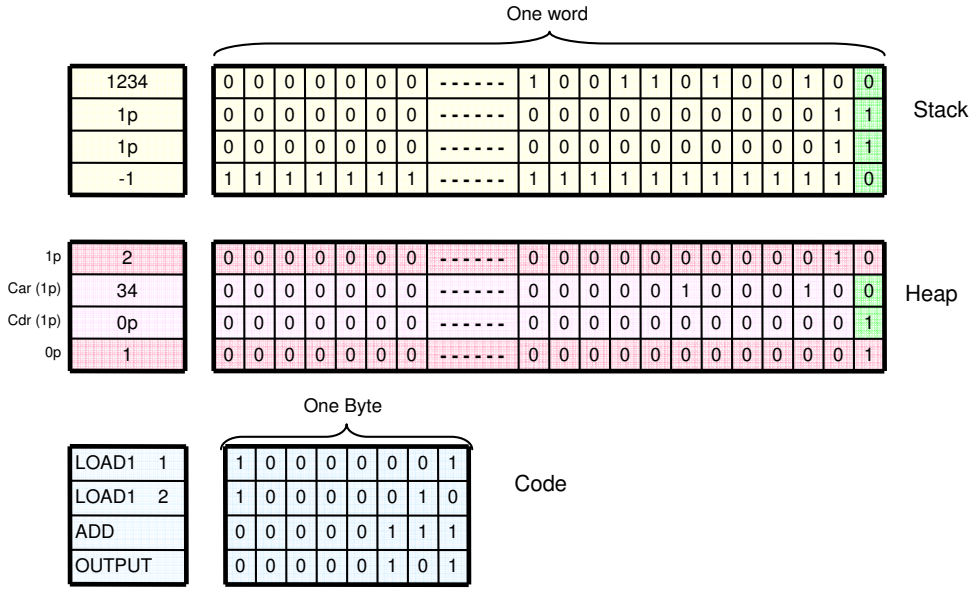

Figure 3.3: Schemata of the stack, the heap, and the code space in the VEP

Figure 3.3 shows schemata of the stack, the heap, and the code space in the VEP. For each of these three schemata, sample binary content are shown on the right-hand side and the human readable format of the same content on the left-hand side. As an example we consider the content of the top element of the stack. Since the type bit (the right-most bit) is zero the content of the cell has the type *number* which is represented by the rest of the bits (1234 in human readable format). In the same way, the second stack element from the top has the type *pair* (typebit is one) and rest of the bits show the address of the pair in the heap which is 1 ($1p$ in human-readable format). The pair $1p$ in the heap is a pair of the two values 34 and $0p$ which are respectively the *car* and the *cdr*[1] of $1p$, where *car* returns the first item of the pair and *cdr* returns the second one. It should be mentioned that the values in the pairs follow the same typing convention as we have in the stack. It is worth mentioning that the content of the address $1p$, which is not visible to the programmer, is the number of references to the pair $1p$ which is used for memory management. The memory management of the VEP will be discussed later in Section 3.2.

[1]Generalised from the LISP operations on binary tree structures, where *cdr* returns a list consisting of all but the first element of its argument and *car* returns the first element.

Instruction Set

The design of the instruction set is one of the most interesting and important aspects of the VEP design. The first step was to choose the instructions we wanted to implement. We start by choosing some generic instruction types. For a first attempt, we limited the instructions to some well-known and common instructions of other machines (processors) in the following categories:

- Arithmetic instructions: the basic four integer arithmetic operations are addition, subtraction, multiplication, and division.
- Logical instructions: these instructions usually work on a bit by bit basis. Typical logical operations include logical negation or logical complement, logical and, logical or.
- Comparison instructions: the compare instructions compare values by using a specific comparison operation. Typical logical operations include equal and not equal.
- Data transfer instructions: these instructions move data from one location in memory to another. Data movement instructions typically come in a variety of sizes depending on machine structure.
- Control instructions: machines and processors, by default, work on instruction sequence. Redirection from this sequence is possible through control instructions. The most basic and common kinds of program control are the unconditional jump and the conditional jumps (branches). Control instructions also include instructions which directly affect the entire machine such as halt or nop.

Rank	80x86 instructions	% Execution
1	Data transfer instructions	38.00%
2	Control instructions	22.00%
3	Comparison instructions	16.00%
4	Arithmetical instructions	13.00%
5	Logical instructions	6.00%
	Total	96.00%

Figure 3.4: Instruction distribution approximation

After gathering an initial set of instructions, we started setting the distribution of the instructions. In order to do this we used the Figure 3.4 as a general guideline. The table is an approximate interpretation of the work of Hennessy *et al* [40] in which 10 simple

instructions that account for 96% of instructions executed for a collection of integer programs running on the popular Intel 80x86 was presented. We also kept an eye on our representative algorithm to find out how frequently some operations are executed.

1	PEEK	9	LOAD4	17	NEQ	25	JMPRF
2	POKE	10	CONS	18	BAND	26	OUTPUT
3	POP	11	CAR	19	BOR	27	HALT
4	PUSH-PC	12	CDR	20	BNOT	28	ADD
5	READC	13	ISPAIR	21	BSHIFT	29	SUB
6	LOAD1	14	EQU	22	JUMP	30	MUL
7	LOAD2	15	LTH	23	JMPR	31	DIV
8	LOAD3	16	LEQ	24	JMPRT	32	MOD

Figure 3.5: Initial instruction set

Figure 3.5 shows the resulted initial instruction set which consists of 32 instructions that would allow programmers to efficiently write programs.

One design choice that we made was about the jumping instructions. Technically, the destination in jumping instructions can be absolute or relative to the program counter. In relative jumps the offset of the destination is found on the top of the stack. In the absolute jumps, the positive interpretation of content of the top of the stack is the effective address of the destination. In relative jumps the offset (which could be positive or negative) is added to the program counter to yield the effective address of the destination. Most of the jumps in our representative algorithms are the loops which can be easier implemented with conditional relative jumps. Therefore, we decided to have three relative jumps (two conditional and one unconditional), and one absolute jump. The absolute jump can be used in the cases where the relative distance is not (easily) available (like jumping to a subroutine or jumping back from a subroutine). This offers a good flexibility to the programmer to use the jump which he finds more suitable for the situation.

One more important aspect about the selected initial instructions is the absence of instructions which operate on network or gives the ability to inspect the host system. Furthermore, there are no instruction which can read from input devices and write into file streams. These are to enable the execution of the proof generator at the consumer side in a secure manner. That is, we tried to enforce security policies such as no access to files or no access to the network on instruction set design level. Thus, the selected instructions provide the VEP with a tightly-controlled environment for proof generator to run in.

Among the 32 initial instructions, 12 (1-12) are data transfer instructions. The PEEK instruction pops p then pushes the content of the stack location specified by p.

The POKE instruction pops[1] p, pops q, then writes q to the stack location specified by the first popped element. READC pops p then pushes the byte found in code space at the position specified by p, and PUSH-PC pushes the current value of the program counter onto the stack. LOAD1, LOAD2, LOAD3, and LOAD4 respectively push the numeric value encoded by the next 1, 2, 3, and 4 byte(s) in the code space onto the stack in big-endian format.[2] The CONS instruction, pops p, pops q, then constructs a new pair containing (q, p), in the heap, and pushes the heap address of the built pair onto the stack. CAR pops p, then pushes the left child of the pair with the heap-address specified by p onto the stack. CDR pops p, then pushes the right child of the pair with the heap-address specified by p onto the stack. Out of the remaining instructions, there are six control (22-27), five arithmetics (28-32), five comparisons (13-17), and four logical (18-21) instructions.

After deciding which instructions to include in our initial instruction set, the next step was to assign *opcodes* for them. An Opcode is the binary representation of one instruction. The encoding of opcodes affects the size of the compiled program. One design choice was to choose between the variable-length encoding and the fixed-length encoding. Simplicity favors regularity which suggests having just a single instruction length with as fixed format as possible across the whole instruction set.

First, we group the instructions into subsets according to their common characteristics. For example, the first four instructions are stack data transfer instructions with no operands (some of them have implicit operand(s) on the stack) so we put them in one group. LOAD1, LOAD2, LOAD3, and LOAD4 are the only instructions which have explicit operand[3], so they are grouped together. CONS, CDR, CAR, ISPAIR are grouped as heap-related instructions. We assigned opcodes to the instructions beginning by assigning the first opcode (000000000) to the first instruction of the first subset and so forth. Figure 3.6 shows the resulted encoding for the initial instruction set.

As we mentioned earlier, the code space of the VEP is stored in byte-length cells. This makes it possible to have an instruction set with 256 instructions. We intend to execute the GUNZip algorithm as the representative algorithm on the VEP. Since data decompression algorithms perform their operations on data, they might execute more efficiently on a machine with rich set of data transfer instructions. Therefore, we use the remaining 224 as data transfer instructions. To do so and as the first step, we

[1] The "push" and "pop" used to explain the instructions are not done incrementally and the stack pointer is set at the end of the execution of the instruction (e.g., the POKE instruction reads the first and the second element of the stack from the top and writes the second element to the stack location specified by the first element then does two consecutive pops).

[2] A format for storage of binary data in which the most significant bit is placed first.

[3] The arguments encoded in after the opcode.

n	Instruction	Opcode	n	Instruction	Opcode
1	PEEK	0000 0000	17	BAND	0001 0000
2	POKE	0000 0001	18	BSHIFT	0001 0001
3	POP	0000 0010	19	BNOT	0001 0010
4	PUSH-PC	0000 0011	20	BOR	0001 0011
5	READC	0000 0100	21	CONS	0001 0100
6	OUTPUT	0000 0101	22	CAR	0001 0101
7	HALT	0000 0110	23	CDR	0001 0110
			24	ISPAIR	0001 0111
8	ADD	0000 0111			
9	SUB	0000 1000	25	JUMP	0001 1000
10	MUL	0000 1001	26	JMPR	0001 1001
11	DIV	0000 1010	27	JMPRF	0001 1010
12	MOD	0000 1011	28	JMPRT	0001 1011
13	EQU	0000 1100	29	LOAD1	0001 1100
14	LTH	0000 1101	30	LOAD2	0001 1101
15	LEQ	0000 1110	31	LOAD3	0001 1110
16	NEQ	0000 1111	32	LOAD4	0001 1111

Figure 3.6: Opcodes of the initial instruction set

provide the programmers with additional versions of PEEK and POKE, PEEK-i and POKE-i which are faster and more compact. Figure 3.7 shows the difference between the two versions of PEEK.

Figure 3.7(a) shows a snapshot of the stack. The indices on the left side next to each cell shows the address of that cell and the stack grows toward up. The highest address is the current value of the stack pointer. We want to copy the content of the middle cell (50) to the top of the stack. Using the PEEK instruction from the initial instruction set, we can perform the task in two different ways. The code fragment in Figure 3.7(b) tells the VEP to copy the content of cell number 1 to the top of the stack. That is, the PEEK instruction pushes the content of the stack location e from the bottom to the stack, where e is the positive topmost popped element of the stack.

The second way to perform the same task is shown in Figure 3.7(c). If the topmost element of the stack is a negative value, the PEEK instruction pushes the content of eth stack location from the top onto the stack. Both of the presented code fragments use more than 1 byte to perform a very frequent task. Figure 3.7(d) shows the additional version of the PEEK instruction, called PEEK-i, which can perform the same task using only one byte.

Assuming that PEEK operation targets a small portion of the top of the stack more frequently, PEEK-i (i from 1 to 8) uses the same semantics as the PEEK instruction (but with no operand on the stack) to push the value of stack-location relative to the stack pointer with offset of i onto the stack. The instruction POKE-i (i from 1 to 8) is designed in the same way to pop the value e and write it to the stack location relative to the stack pointer with offset i. Therefore, PEEK-i is intended to act like "LOAD1 -i" followed by a PEEK. Likewise, POKE-i is intended to act like "LOAD1 -i" followed by a POKE. Considering the frequency of the copying and moving tasks, these additional instruction can help us to have a smaller size of code for the proof generators.

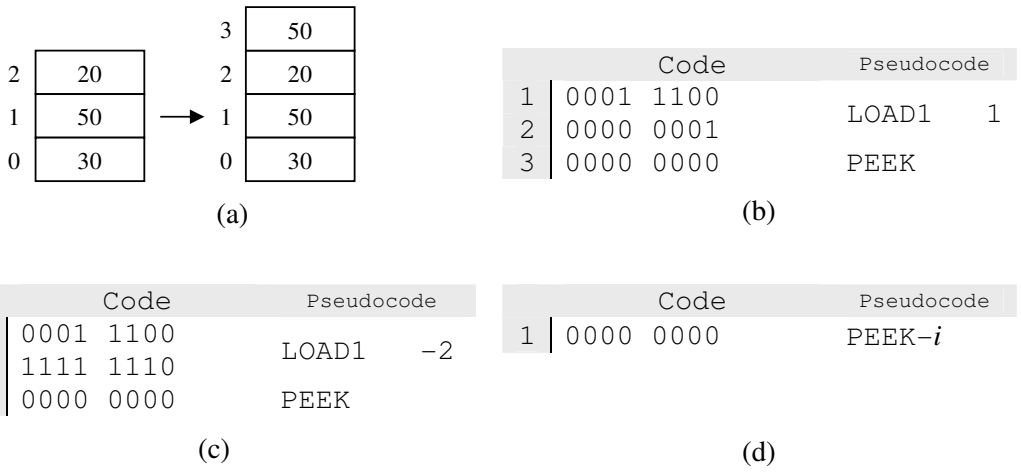

Figure 3.7: Example of PEEK-i

POKE-1 is an interesting case of POKE-i. Figure 3.8 shows an example of using POKE-4 and compares it with POKE. In this figure, we can see the code fragment on the left side and, in front of it, the stack snapshots corresponding to each instruction of the code fragment. Now suppose that we have the code fragment of Figure 3.8(a). In this figure, in stack snapshot I, the stack pointer (SP) is equal to 3. The programmer does LOAD1 17 and reaches the stack snapshot II. Then he performs POKE-1 and gets to snapshot III (SP=5). By doing POKE, the content of the stack position SP+(-1) (equal to 4) is overwritten by 17 and SP is set to 3. Therefore, if we compare the snapshot IV with snapshot I the programmer has not changed the stack with his code fragment because the value 17 is overwritten on itself and then popped. As we can see in Figure 3.8(d), POKE-1 performs exactly the same. By comparing the snapshot II with III in this figure one can easily conclude that the net effect of doing POKE-1 is the same as doing POP. Therefore, the instruction POP can be assigned to the opcode of POKE-1. In this way we can take advantage of the availability of the previous opcode of the POP, and assign it to NOP (no operation) instruction.

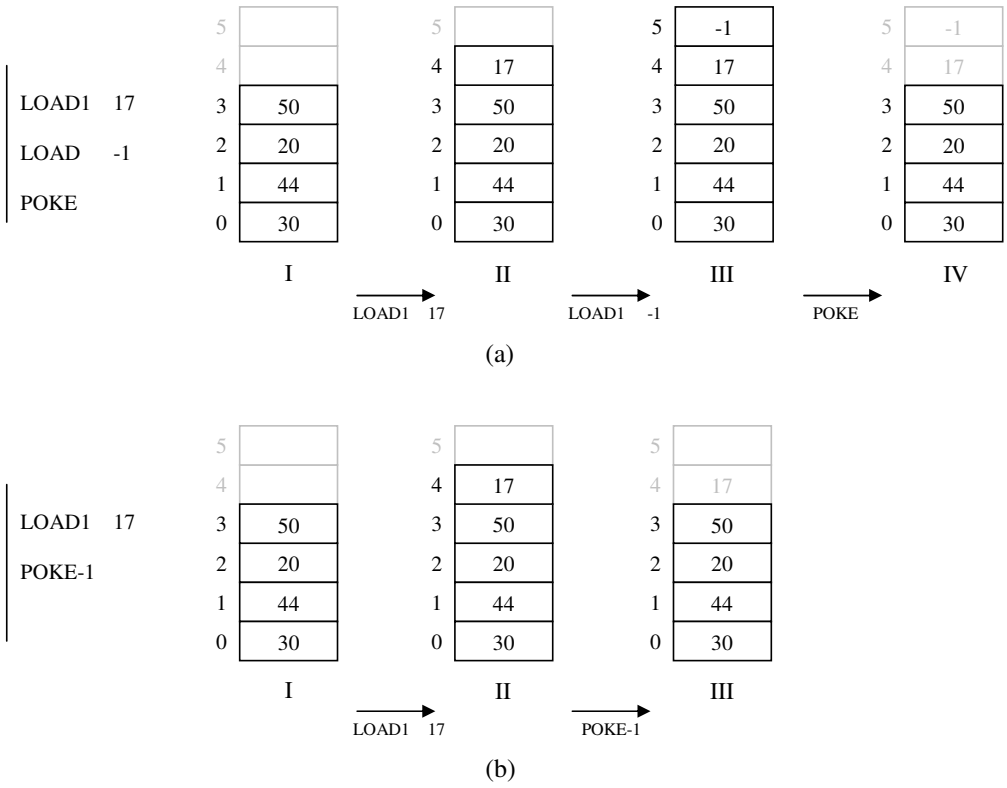

Figure 3.8: Example of POKE-i

The instruction LOAD1, LOAD2, LOAD3, and LOAD4 are the only instruction which have explicit operand. Considering the frequent usage of LOAD1, we design LOADi as the partial shorter version of LOAD1 and assign the rest of available opcodes (00110000 to 11111111) to the instructions in LOADi group. Figure 3.9 shows how these opcodes are distributed in details. We refer interested readers to Appendix A for a detailed description of each instruction.

In instructions PEEK-i, POKE-i, LOADi, the first few bits can be regarded as the actual opcode and the following bits as the immediate operand. In this way, we can consider the newly added instructions as instructions with shorter format than the other instruction which leads to higher code density.

In instructions PEEK-i, POKE-i, LOADi, the first few bits can be regarded as the actual opcode and the following bits as the immediate operand. In this way, we can consider the newly added instructions as instructions with shorter format than the other instruction which leads to higher code density.

n	Instruction	Opcode	n	Instruction	Opcode
1	PEEK	0000 0000	25	JUMP	0001 1000
2	POKE	0000 0001	26	JMPR	0001 1001
3	NOP	0000 0010	27	JMPRF	0001 1010
4	PUSH-PC	0000 0011	28	JMPRT	0001 1011
5	READC	0000 0100	29	LOAD1	0001 1100
6	OUTPUT	0000 0101	30	LOAD2	0001 1101
7	HALT	0000 0110	31	LOAD3	0001 1110
			32	LOAD4	0001 1111
8	ADD	0000 0111			
9	SUB	0000 1000	33	PEEK-1	0010 0000
10	MUL	0000 1001	34	PEEK-2	0010 0001
11	DIV	0000 1010	35	PEEK-3	0010 0010
12	MOD	0000 1011	36	PEEK-4	0010 0011
			37	PEEK-5	0010 0100
13	EQU	0000 1100	38	PEEK-6	0010 0101
14	LTH	0000 1101	39	PEEK-7	0010 0110
15	LEQ	0000 1110	40	PEEK-8	0010 0111
16	NEQ	0000 1111			
			41	POP	0010 1000
17	BAND	0001 0000	42	POKE-2	0010 1001
18	BSHIFT	0001 0001	43	POKE-3	0010 1010
19	BNOT	0001 0010	44	POKE-4	0010 1011
20	BOR	0001 0011	45	POKE-5	0010 1100
			46	POKE-6	0010 1101
21	CONS	0001 0100	47	POKE-7	0010 1110
22	CAR	0001 0101	48	POKE-8	0010 1111
23	CDR	0001 0110			
24	ISPAIR	0001 0111	49 - 176	LOADi (0 ... 127)	1xxx xxxx
			177 - 240	LOADi (-1 ... -64)	01xx xxxx
			241 - 256	LOADi (-65 ... -81)	0011 xxxx

Figure 3.9: Complete instructions set: instructions and their corresponding opcodes

3.2 Memory Management

In order to manage the heap memory, the VEP should provide the programmers with certain means of allocating the heap memory spaces and reclaiming them when they are no longer needed.

In *traditional memory management* the programmer manages the memory, using available operations for allocation and de-allocation. In order to correctly handle the memory management in traditional memory management, the programmer should put a large portion of his time on low level memory operations or face severe problems such as *dangling references*. Dangling references are the result of wrong (premature) de-allocation of an object while a valid reference to de-allocated object still exists. If the programmer is lucky enough, this can lead to a crash of the program, and if not, the program may continue execution and generate false results. Another possible problem is memory leak due to dropping the last reference to an object without freeing the no-longer referred object. In this way, the memory leak may lead to exhaustion of memory.

On the other hand, *garbage collection* or *automatic memory management* makes life easier for programmer. In automatic memory management, the garbage collector allocates memory in response to memory requests and reclaims unused memory automatically. The VEP uses automatic memory management, thus there would be no dangling reference or memory leak due to manual memory management errors and the programmer can put more time on productivity instead of managing low-level memory operations.

Reference counting is the garbage collection technique used in the VEP. In reference counting technique, each object (pair) in the heap contains a counter which tracks the number of references to that object (references from stack elements and heap objects). That is, the reference-count field of the object is incremented when there is a new reference to that object, and it is decremented when the reference is removed. Figure 3.10 (a) shows an example of this method where the reference a in the top figure is dropped which results in the figure at the bottom. When the reference count falls to zero, there are no more references to the object. Therefore, we can immediately find unreachable objects, and then reclaim them.

To be sure that all objects will have a correct reference-count during the execution of the program, the following issues should be taken into consideration:

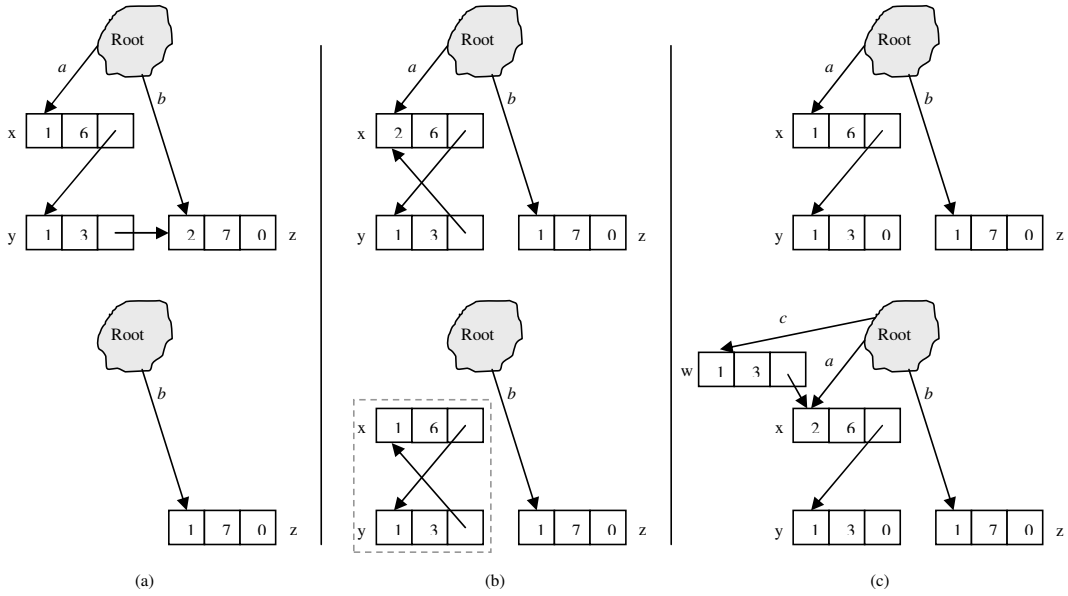

Figure 3.10: Reference counting examples

- When the reference count falls to zero, the reference counts of offspring objects should be decremented before the object is reclaimed (Figure 3.10 (a)). Decrementing offspring objects before removing the last pointer of the object is done by a simple recursive algorithm shown in Figure 3.11.

- Care should be taken in reference counter increment to avoid an overflow which would make the counter distrustful. This will happen if the reference counter is not large enough. By choosing a word-sized reference counter which is as large as the maximum number of references the memory can hold the problem is solved.

A major drawback of reference counting is its failure in reclaiming cyclic garbage data structures which is shown in Figure 3.10 (b). If a cyclic data structure becomes inaccessible from the root set, its constitutive objects keep each other's reference count greater than zero. Thus, pairs x and y are not reclaimed.

In Figure 3.10 (c), we want to update the pair y to make it point to the pair x. The figure at the bottom shows the result. Notice that the pair y has been copied as w, but the pair x is shared. As a result, the original pairs (x and y) persist and have not been modified. The reason for the copy is that the right-hand child in y (the cell containing the original value 0) cannot be modified to point to the pair x. Since every value in in the VEP is built up out of existing values, it is impossible to create a cycle of references, resulting in a reference graph (a graph which has edges from objects to the objects they reference) that is a directed acyclic graph. Therefore, the reference counting in the VEP does not have the drawback of failure in reclaiming cyclic data

structures.

```
void ifpair_reduce(vmword wrapped_data)
{
  if (wrapped_data & 1)                    //is it a pair?
       {
       vm_u_word pair_adrs;
       pair_adrs=wrapped_data>>1;          //extract the pair address
       h[pair_adrs]-=1;                    //reduce the reference count
       if (!(h[pair_adrs]))                //reference count is zero?
           {
           ifpair_reduce(h[pair_adrs-1]);  //decrease the reference count of car
           ifpair_reduce(h[pair_adrs-2]);  //decrease the reference count of cdr
           addtofree(usint);               //reclaim the object
           }
       }
}
```

Figure 3.11: The top element of the stack (wrapped_data) is passed to the recursive algorithm (ifpair_reduce) before popping. If the top element is a pair, by popping we are actually releasing a reference to that pair, so the counter field of that pair should be decreased by one. Furthermore, if the reference count falls to zero, the reference counts of children objects should be decremented before the object is reclaimed

3.3 Security Requirements

We designed the VEP such that it guarantees a certain number of fundamental safety properties in order to execute the untrusted code in a secure manner. The following fundamental safety properties are the security requirements of the VEP.

- *Type safety* verifies that a code is well-typed according to a type system defined for the language. That is, the operations are applied only to operands with appropriate types.
- *Numeric safety* checks that programs will perform arithmetic correctly. In other words, the arithmetic instructions should have legal arguments. This security requirement is to avoid potential error of using partial operators with arguments outside their defined domain (e.g., division by zero).
- *Memory safety* prevents reading and writing to illegal memory locations. Technically, one can read the code space and performs read and write in the heap and the stack space. The legal code space locations to read from are the locations with their address less than the code size and greater than or equal to zero. Even the instruction loading must be performed as legal reads from the code space. The only way to write a piece of data in the heap is to build a pair which has the

data as its children. Since the only way to construct a pair is using the instruction CONS, all the pairs are legally built by the VEP and have their type bit 1. Likewise, in order to read (access) any data in the heap, it is enough to perform the CAR and CDR instructions on a pair. Since the construction of the pairs is governed by the VEP, and the programmer has no mean to modify the type bit to build a new pair, it is enough to perform a type check before the CAR and CDR instruction to be sure that they are going to be performed on a pair. On the other hand, any read and write to the stack should be preceded by a memory check which assures that the reading and writing are going to be performed on valid stack locations as their destination. The validity of the destination varies from instruction to instruction. Generally, the valid reading destinations are stack locations with their address less than or equal to the stack pointer and greater or equal to the stack base. The valid writing destinations are the same as reading ones, except that the location above the top of the stack is valid for some instructions to write in. Furthermore, memory safety in the VEP asserts that each operation has a sufficient amount of required memory (stack and heap) to perform the instruction (e.g., the VEP raises an error if an attempts is made to pop when the stack is empty or to push an item onto a stack which has no room for further items).

- *Control flow safety* prevents arbitrary jumps in code. Namely, the program should never execute a jump to a random location, but only addresses within its own code space.

- *Resource bound check*, verifies the requested amount of code, stack, heap space, and the number of executed operations. That is, it checks that the required resources for the code execution are within the bound set, beforehand, in an agreement between the code producer and the consumer. Resource bound check is done on the requested amount of code, stack, heap and the number of executed operations that producer inserts in the code header.

- *Exception handling* properties ensure that all exceptions that can be thrown within a code can be handled and the VEP has the ability to deal with errors automatically.

3.4 Security Enforcement

The security enforcement by the VEP is simple and straightforward. The VEP enforces the security requirements at different levels. Categorizing the security checks according

to their enforcement level shows better how easy the VEP security enforcement is to perform and understand.

3.4.1 Initial security enforcement

The VEP checks the following requests, only once, upon receiving the untrusted code before executing the code:

- *Code size*: the producer inserts the demanded code size (*dcs*) of the proof generator in the header of the untrusted code. This demanded code size, then, is checked by the VEP to assign necessary code space to the code. The size of the code can be talked over in an agreement in which the producer and the consumer settle the possible amount of the resources that can be used by the untrusted code. The VEP allocates a block of *dcs* bytes of memory as the code space and inserts the code into the code space. If the VEP fails to allocate the requested block of memory, or the actual size of the code is greater than the demanded one, the VEP refuses the untrusted code.

- *Stack size*: The VEP also checks the demanded stack size (*dss*) in the header of the untrusted code, requested by the producer, to assign necessary stack memory to the code. The VEP allocates a block of *dss* words of memory as the stack memory. If the VEP fails to allocate the requested block of memory, the VEP refuses the untrusted code.

- *Heap size*: the amount of demanded heap size (*dhs*) of the untrusted code is also mentioned in the code header. The VEP assigns the necessary heap memory to the untrusted code. Then, the untrusted code can make allocation in this heap memory, without having the responsibility of deallocation (this is done by the automatic memory management of the VEP). The heap size can also be settled, beforehand, in the agreement between the producer and the consumer. The VEP allocates a block of *dhs* words of memory as the heap memory. If the VEP fails to allocate the requested block of memory, the VEP refuses the proof generator.

- *Operation number*: the untrusted code should finish its task within a definite time period which could also be agreed beforehand for productivity reasons. The number of operations which is inserted by the code producer into the code header is checked by the VEP not to be more than a certain limit. In the case where the requested number of operations is more than the limit, the VEP refuses the proof generator.

3.4.2 Global security enforcement

After the initial security enforcement, if the code was not refused, it is ready to be executed in the VEP. Throughout the execution, the VEP enforces two security checks globally. That is, these two checks are independent of the actual next instruction that is aboutto be executed. The global security enforcement consists of checking the followings.

- *Execution time*: before fetching the next instruction, the VEP makes sure that the code execution time (measured by the number executed operations) has not exceeded the requested number of operations which was approved in initial security enforcement. If the number of executed operations is less than the approved number, then the check is passed, otherwise the code execution will be refused.
- *Program counter*: The VEP should check if the program counter points inside the code space (i.e., non-negative and less than the code size).

3.4.3 Instruction-wise security enforcement

The third level of security enforcement by the VEP is the fine-grained level and done per instruction. By enforcing this level of security checks the untrusted code is prevented to perform any unsafe operation. Therefore, after fetching the instruction and before the execution of the instruction, the VEP performs a combination of the following checks.

- *Number of operands*: the number of operands in the VEP instruction set can vary from zero to two implicit operands on the stack, depending on the instruction (e.g., ADD requires two and POP requires one operand(s) on the stack). For the instructions with one or more operands on the stack, the existence of a sufficient number of operands should be checked before execution of the instruction. If the sufficient number of operands for an instruction is not available, the execution is discontinued and the untrusted code gets refused.
- *Type of operand*: the VEP checks if the type of operands conforms with the operation. As mentioned earlier, the operands in the VEP can be numbers or pairs. The VEP can distinguish the type of an operand according to its type bit. The only instructions that should have pair-type operand are CAR and CDR. The type of the operands of the EQU, NEQ, POP, ISPAIR, and POKE-i instructions is not important because these instructions perform their operation on the content

of one or more locations of the stack (and not the contents of one or more cells of the stack). The other instruction which have the leftmost column checked in Figure 3.13 are the instruction which should have number-type operand(s). Checking the type of operands ensures that a code is well-typed according to the VEP's type system. That is, the operations are applied only to operands with correct types.

- *Legal range of operands*: the arithmetic instructions should have legal arguments. The VEP checks the operand legality to prevent potential error of using partial operators with arguments outside their defined domain (e.g., division by zero).
- *Legal stack destination*: For any instruction which results in a read or write to the stack VEP assures that the reading and writing have legal stack locations as their destination.
- *Sufficient memory*: The VEP verifies the minimum amount of required memory (stack and heap) to perform the instruction which work with memory.
- *Legal code destination*: before changing the program counter to the jump destination, the VEP checks if the destination is within the code space. LOAD1, LOAD2, LOAD3, and LOAD4 are the only instructions with the next 1, 2, 3, and 4 byte(s) encoded as their operand. Therefore, the VEP checks if fetching their operand dose not cause reading beyond the code size. It should be mentioned that the VEP does not enforce the concept of instruction boundaries (e.g., the VEP accepts a jump in middle of a LOAD4 instruction). Therefore, the VEP can accept intertwined codes. Figure 3.12 shows an example of an intertwined code, where each column represents a particular byte and that byte is the one given by the instruction in that column. In the example, the control can jump in any of the addresses A to A+4 and execution would proceed without error. In each case a different constant would have been pushed.

A	A+1	A+2	A+3	A+4
LOAD4	operand	operand	operand	operand
	LOAD3	operand	operand	operand
		LOAD2	operand	operand
			LOAD1	operand
				LOAD*i* 1

Figure 3.12: An example of an intertwined code

As it is shown in Figure 3.13, the complete set of instructions with their safety checks can be simply put into a table. In this way it would be an easy task to verify the safety of the VEP by pen and paper.

Ins	T	R	N	S	C	M
PEEK	✓		✓	✓		
POKE	✓		✓	✓		
POP			✓			
PUSH-PC						✓
READC	✓		✓		✓	
OUTPUT	✓	✓	✓			
HALT						
ADD	✓		✓			
SUB	✓		✓			
MUL	✓		✓			
DIV	✓	✓	✓			
MOD	✓	✓	✓			
EQU			✓			
LTH	✓		✓			
LEQ	✓		✓			
NEQ			✓			
BAND	✓		✓			
BSHIFT	✓		✓			
BNOT	✓		✓			
BOR	✓		✓			
CONS			✓			✓
CAR	✓		✓			
CDR	✓		✓			
ISPAIR			✓			
NOP						
JUMP	✓		✓		✓	
JMPR	✓		✓		✓	
JMPRF	✓		✓		✓	
JMPRT	✓		✓		✓	
LOAD(1..4)					✓	✓
PEEK-*i*				✓		✓
POKE-*i*			✓	✓		
LOAD*i*						✓

T: Type of operands check
R: Range of operands check
N: Number of operands check
S: Stack destination check
C: Code destination check
M: Memory sufficiency check

Figure 3.13: Instruction-wise security enforcement

3.4.4 Example

Figure 3.14 shows the first ten lines of two codes. The only difference between these two code fragments is in their last instruction. We assume that both of these codes have passed the initial level of security enforcement. Figure 3.15 shows the resulted stack after executing each of these codes up to the eighth instruction (the number of performed operations is 7 and the program counter is equal to 7, next the ADD is going to be executed). Thus, we are going to discuss the safety of the last three instructions.

Before fetching the next instruction, the VEP checks if the execution time is not beyond the limit and the program counter is less than the code size. Supposing that the requested execution time for these codes is 20 operations and the code size is 100 bytes, $7 < 20$ and $7 < 100$ so these two checks are passed). Then, the VEP fetches the next instruction which is ADD. The ADD instruction needs two numeric arguments. Therefore, the VEP first checks if there exist at least two elements on the stack. Here we have 5 elements on the stack, so the check is passed. Then, it checks if the type of these two elements is *number*. Here, the type bit of the top two elements is zero which indicates the type *number*. Then, the VEP pops 2 and 43 and pushes 45 with a zero type bit.

	Code (a)	Pseudocode
1	0100 0000	LOADi-1
2	1000 1100	LOADi12
3	1100 0111	LOADi71
4	0001 0100	CONS
5	1000 0011	LOADi3
6	1010 1011	LOADi43
7	1000 0010	LOADi2
8	0000 0111	ADD
9	0001 0100	CONS
10	0001 0110	CDR

	Code (b)	Pseudocode
1	0100 0000	LOADi-1
2	1000 1100	LOADi12
3	1100 0111	LOADi71
4	0001 0100	CONS
5	1000 0011	LOADi3
6	1010 1011	LOADi43
7	1000 0010	LOADi2
8	0000 0111	ADD
9	0001 0100	CONS
10	0000 1000	SUB

Figure 3.14: Example codes

Before the next instruction, the VEP performs the first two checks of *execution time* and *program counter* as performed for ADD instruction ($8 < 20$ and $8 < 100$ so these two checks are passed).

Then, the VEP fetches the CONS instruction. The CONS instruction constructs a pair out of two values without any type restrictions. Therefore, the VEP checks if there exist at least two elements on the stack. Here we have 4 elements on the stack, so the check is passed. Then, the VEP pops 45 and 3 and builds a pair, in the heap, with 3 as its first child and 45 as the second child (it should be mentioned that the types of children are preserved in building the pairs). Supposing that the newly allocated pair in the heap has the address 1, the VEP pushes 1 onto the stack and sets the type-bit to one which shows the type *pair* (shown as $1p$ in human readable format).

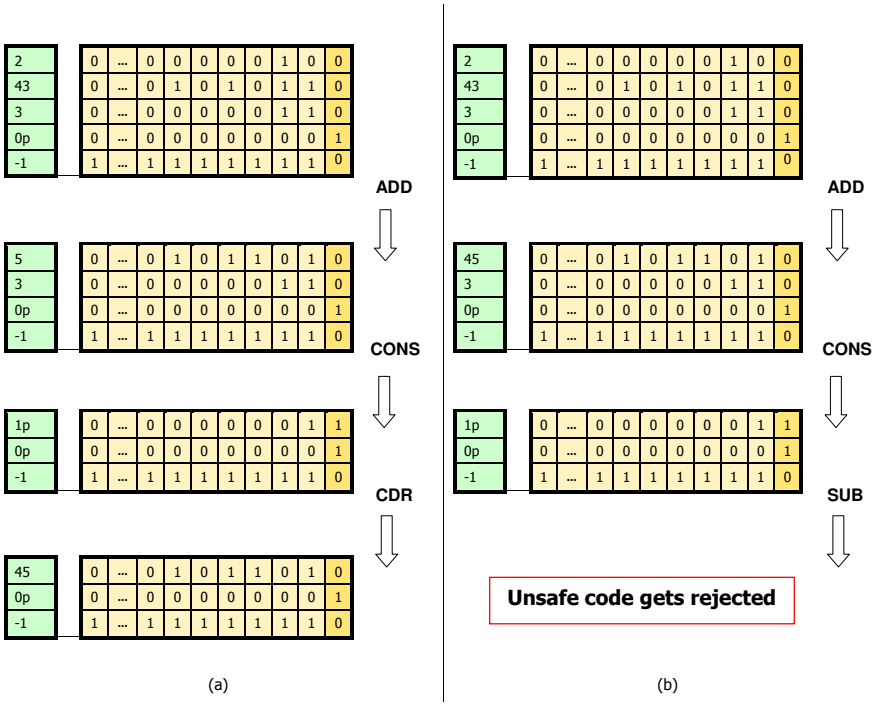

Figure 3.15: Stack schemata of the example

Before the next instruction, the VEP performs the first two checks of *execution time* and *program counter* ($9 < 20$ and $9 < 100$ so these two checks are passed). For code of Figure 3.14(a), the VEP fetches the CDR. The CDR instruction receives a pair and returns the second child of the pair. Therefore, the VEP checks if there exist at least one element on the stack. Here, we have 3 elements on the stack, so the check is passed. Then, the VEP checks if the type of the element is *pair*. Here, the top element ($1p$) is a *pair*. Then, it pops the pair $1p$ and pushes 45 on the stack.

For code of Figure 3.14(b), the VEP fetches SUB. The SUB instruction needs two numeric arguments. Therefore, the VEP first checks if there exist at least two elements on the stack. Here, we have 3 elements on the stack, so the check is passed. Then, it checks if the type of these two elements is *number*. Here, the type bit of the top element is one which indicates the type *pair*. Thus, the check fails and the VEP rejects the unsafe code.

3.5 The VEP Versus Other VMs

The EPCC framework uses the VEP in order to execute untrusted proof generators at the consumer side in a secure manner. There are many systems that execute untrusted

codes in virtual machines to limit their access to system resources. Therefore, a question one could ask is "why not use an other existing virtual machine instead of the VEP?". Here, we try to highlight the main reasons of choosing the VEP over two best-known virtual machines. These two virtual machines are: Java virtual machine (JVM) [47] introduced in 1995 by Sun, and the .NET platform [48] developed more recently by Microsoft.

Any virtual machine that we choose would be a part of the TCB in EPCC framework. Knowing that any bug in TCB can compromise the security of the whole system, we should choose a virtual machine which increases the size of the TCB the least (as required by the second principle of security design). Using either JVM or .NET results in a large TCB, potential source of allowing many possible points of failure (these large TCBs were the motivations for introducing the PCC approach in the first place). Appel *et al.* [49] measured the TCBs of various Java virtual machines at between 50,000 and 200,000 lines of code. Figure 3.16 shows the TCB size comparison between these JVMs and the VEP. The TCB size in these two JMVs is even bigger than the TCB size of the traditional PCC. Therefore, using these virtual machines to extend the PCC framework results in an undesirably huge TCB and ineffectual PCC framework.

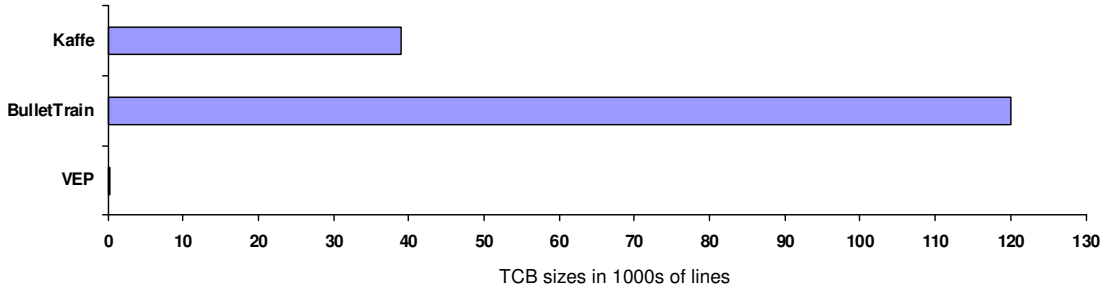

Figure 3.16: The TCB sizes of various JVMs (in thousands of lines of code): Kaffe is an open-source nonoptimizing Java JIT, BulletTrain is a highly optimizing Java compiler.

For EPCC, we need a virtual machine with a simplicity, such that, it be feasible for a human to inspect and verify it by pen and paper. None of the mentioned virtual machines and any other that we are aware of have been developed with this goal. JVM, .NET, and other best-known virtual machines are mostly focused on the performance, portability, etc. The VEP is developed in less than 300 lines of code which makes it possible to be easily verified by human and gives it the potential of being proven safe in future. Therefore, we have shown that the VEP is orders of magnitude smaller, and simpler than popular virtual machines.

Chapter 4

Sample Use of EPCC

In this chapter, we present a sample implementation of an EPCC framework. As shown in Figure 4.1, we intend to extend the PCC framework by employing EPCC. In this way, instead of sending the proofs to the consumer, a proof generator can be sent, which then can get executed safely on the VEP. In Chapter 3, we discussed the design of the VEP in details. The only other new component to this framework, in comparison to the traditional PCC framework, is the *proof generator builder* which outputs a VEP executable proof generator. Since it would be very tedious to write programs in the VEP machine language, we need an assembler to enable a programmer to use instructions instead of opcodes. Writing the proof generator program in a high-level language is even more favorable. Thus, we implemented an *assembler* to generate the machine code for the VEP, a *C compiler* which generates the assembly code for the assembler, and a *proof generator program* in C language. Next, we talk about these three components, then, we present an overall view and the detailed diagram of the implemented framework.

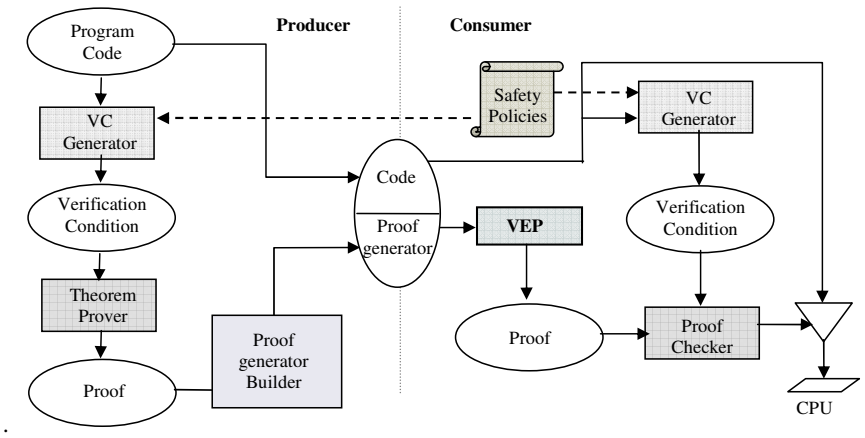

Figure 4.1: Extending the PCC framework

4.1 Assembler

An assembly language is a type of low-level computer programming language consisting mostly of symbolic equivalents of a particular machine language. Machine languages consist entirely of opcodes and are almost impossible for humans to read and write. Assembler languages (we will also refer to it as assembly languages) have the same structure and set of commands as machine languages, but they enable a programmer to use instructions instead of opcodes and to organize programs in a readable way. Each type of target machine has its own machine language and assembly language, so a program in assembly language written for our virtual machine, likely, will not run on another machine.

This section describes the language in which the source files must be written. The VEP assembler translates source files in the assembly language format specified in this document into the VEP virtual machine language. We also describe the syntax of the VEP assembly language. The VEP assembler requires a source file in assembly language format as its input which must have a certain structure and content. The specification of this structure and content constitutes the syntax of the assembly language. A source file may be produced by a programmer using a text editor. One other way of producing the assembly source file is using a compiler which translates the codes written in a high-level language to the assembly language. Next, we describe the components of the VEP assembly language.

4.1.1 Statements

The input source code is a text file consisting of a sequence of statements. Each statement may end with the first occurrence of a space or a newline character (ASCII `LF`). Thus, it is possible to have several statements on one line. If several statements appear on a line, they must be separated by spaces (blank character) and the assembler places no restrictions on the number of white space characters between the statements.

Any statement remains a statement even if it is prefixed by a label. Thus, we consider a label as the first field of a statement. Each statement in the VEP assembly language may include up to three fields: a label, an operation (instruction mnemonic or assembler directive) and an operand. As we mentioned earlier, for the case of having several statements in one line, they should be delimited by spaces. The same holds for the statement's fields and their items. That is, the fields of a statement are also delimited by spaces and the items in the fields must also be separated at least one space.

We may use more spaces freely to improve the appearance and readability of a source file without changing the meaning of the statement.

4.1.2 Label field

A label is an identifier which is assigned the immediately following instruction's address and may be referred to throughout the source code. When an identifier is used to label a statement, we speak of a *defining occurrence* of a label, and when it is used to constitute an operand, we speak of an *applied occurrence* of a label. A label should respect the following conditions:

- When it is necessary to refer to a statement in the code, the statement may be labeled with an introductory distinctive identifier of the programmer's choice (as in `EVEN: load loop`). Therefore, a label must be unique, we can not have two or more defining occurrence labels with the same name in the assembly code, but we can have multiple applied occurrences of labels with the same name.
- A label may consist of letters (`A-Z` or `a-z`), numbers (`0-9`), or an underscore (_). The labels are case sensitive (i.e. the upper and lower case of letters are not considered equivalent). Thus `'FOO'` is a different label from `'Foo'`.
- Every label must begin with a letter and not a number.
- Labels may be of any length, but only the first 32 characters are significant. The length 32 is a convention which could be changed later.
- A label, on its defining occurrence, must be terminated by a colon (:) followed by a space or a newline character.

4.1.3 Operation field

The operation field is the second field of a statement. If the statement is labeled (the label field is not blank), the operation field should be preceded by at least one space or newline character. The operations field must be terminated by a space or return. This field must contain one of the followings:

1. A *machine operation*, which is a mnemonic representation of an executable machine language instruction to which it is translated by the assembler. We have two

groups of instruction mnemonic. The members of the first group are the instructions that should be followed by an operand (the operand field should not be left blank). These instructions may occupy more than one byte (as in `load` *`value`*). The second group of instruction are the ones that stand alone and occupy only one byte (like `equ` and `sub`). The instructions of the assembly language described in this document are not in one-to-one correspondence with the VEP instruction set, we refer interested reader to Appendix A for the list of operations in the VEP assembly language and their corresponding machine language instructions.

2. A *pseudo-operation*, which is a directive to the assembler that does not generate any code. It consists of a pseudo-instruction followed by an operand. The current version of the VEP assembly language has only one pseudo instruction `data` which tells the assembler that the following byte is a one-byte data (`data 134`).

4.1.4 Operand field

The operand field provides any data or address information which may be required by the instruction. This field may or may not be required, depending on the instruction. Operands are always mathematical expressions which can include constants and identifiers (labels) and should result in an integer value. The operand field can contain spaces and should be terminated with a space or return character followed by a label or an operation.

Expressions

Some instructions require that the operand supply further data or information in the form of an expression. These expressions may be made up of one or more items and arithmetic operators in standard infix notation, similar in syntax to those of the C language. The expressions in the VEP assembly language are evaluated as full 32bit operations[1]. For some instructions such as `peeki` or `pokei` the result of the expression should be within a definite range (out of this range raises an error) but for `load` it could be from 8 bits to 32 bits and the extra bits will be ignored by the assembler and the assembler truncates the upper half. If truncation occurs an appropriate message will be issued. The item or items used in an expression may be any of two types as

[1] The programmers should be aware that in the VEP architecture the stack and heap cells have only 31 bits. Although a `load` instruction with a four-bytes operand will be successfully translated to the VEP machine language, the left-most bit of the operand will be ignored by the VEP.

listed below. These two types of items may stand alone or may be intermixed by using operators.

1. *Numerical constants*: numbers may be supplied to the assembler as integers in decimal number base.

2. *Identifiers*: labels that are defined assigned some address may be used as identifiers in expressions. As described above under the label field, a label is comprised of letters, digits, and hyphens beginning with a letter. The label may be of any length, but only the first 32 characters are significant. Any label used in the operand field must be defined somewhere in the program.

As mentioned previously, the expressions in the VEP assembly language are mathematical. Therefore, the operators are arithmetical. These operators permit *assembly time* operations such as addition or division to take place. Assembly time is the elapsed time taken for the execution of an assembler. Thus, the expressions are evaluated during the assembly and the results become permanent parts of the code. That is, none of the operations mentioned in the expressions are to be performed by the VEP. The arithmetic operators in the VEP assembly language are shown in Figure 4.2.

Operator	Meaning
+	Binary addition
-	Binary subtraction
*	Multiplication
/	Division (any remainder is discarded)
%	Modulo
^	Power

Figure 4.2: The arithmetic operators in the VEP assembly language

Among these operators, certain take precedence over others in an expression. Power has the highest precedence followed by Multiplication, Division, and Modulo. Addition and Subtraction are the ones with the lowest precedence. All of these precedences can be overcame by the use of parentheses. It should be mentioned that, if there is more than one operator of the same priority level and no parentheses to indicate the order in which they should be evaluated, then the operations are carried out in a left to right order. Figure 4.3 shows some examples of expression evaluation during the assembly time.

Figure 4.4 shows the syntactic rules of expression. The non terminals are represented by lowercase letters, the terminal symbols are represented by uppercase letters, and

Operations like:	Become:
load 4+7	**load** 11
load 4*7	**load** 28
load 4*7+6*2	**load** 40
load 4*(7+6)*2	**load** 104

Figure 4.3: Example of expression evaluation during assembly time

the symbols enclosed in double quotes are terminal symbols. The terminal nodes are given by the last two regular expressions, in which the choices are enclosed in square parentheses. A range of choices is indicated by letters or numbers separated by a dash (-). The asterisk (*) indicates zero or more instances of the previous character.

```
expr     :  addex
         |  expr "+" addex
         |  expr "-" addex
addex    :  mulex
         |  addex "*" mulex
         |  addex "/" mulex
         |  addex "%" mulex
mulex    :  term
         |  term  "^" mulex
term     :  id
         |  number
         |  "(" expr ")"
id       :  LABEL
number   :  DEC-VAL


LABEL    =  [a-zA-Z_][a-zA-Z0-9_]*
DEC-VAL  =  [0-9][0-9]*
```

Figure 4.4: Syntactical rules for expressions

4.1.5 Assembler process

Given an assembly source code with the structure described above, the assembler starts its work by doing passes through the code. In the first pass the assembler scans and parses the code and builds up a symbol table by collecting all symbol definitions. The assembler finds all labels, operations and operands and checks if the code is syntactically correct, that is it verifies that there is no repeated label or malformed statement. Each operand is an expression made of numerical constants, labels, and operators. Thus, the actual value of an operand is a function of the label within that operand. On the other hand, the actual value of a label (position of the label) depends on the size of statements (i.e., size of operations and operands) before that label. In this way, the label positions and the operand values are interdependent. This interdependency makes it impossible to evaluate the label positions and the value of the operands in one pass. In the first pass, the assembler makes an approximation of the position of the labels

in the following way. The position of each label is the summation of the size of the operations and operands before that label where the default size of an operand before calculating its actual value is zero byte and the size of the operations is one byte.

In the second pass, having the approximate label positions, the assembler starts evaluating the value of the operands. If the new size of an operand is more than the its default size (zero byte), the address of the following labels are updated. Updating the label locations, itself, may result in a size change in the operands which have that label as an item. Therefore, the assembler should have a clear policy for the operand size adaptation. Before we explain the policy, we mention the possible operand size changes. Technically, in this process an operand can swell or shrink. Figure 4.5 shows the two possible size change that can occur.

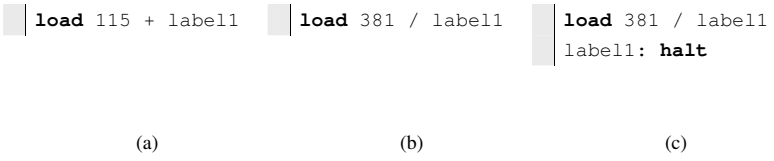

Figure 4.5: Examples of operand size change

In Figure 4.5(a), if the label position is more than 22 the size of the operand grows (e.g., if the label position is 23 the operand evaluation results in `load 128` which translates to LOAD1 128 instead of LOADi). In Figure 4.5(b), depending on the position of `label1` the operand can shrink. For example, suppose that the approximate position of the `label1` was 2, so the operand was evaluated to 190 (`load 190` translates to LOAD2 190). Then, because of an operand size change somewhere before the defining occurrence of the `label1`, it gets updated to position 3. This brings about the operand reevaluation which results in `load 127` which translates to LOADi.

The operand size adaptation have to be strictly increasing. That is, even if an operand shrinks, the assembler has to stick to the size it last used. Otherwise, the problem of finding the most compact selection is NP-hard. Figure 4.5(c) shows an example of the problem, where the `load 381/label1` with `label1` approximated as 1, is evaluated as `load 381` which results in `label1` position update to 3. Then, the reevaluation of `load 381/label1` results in `load 127`. Now, if the assembler decrements the size of the operand, the position of `label1` goes back to 1 and the same scenario happens again. Therefore, the assembler does not decrement any adapted operand size.

Finally, the assembler performs the translation. All the mnemonics and the operand values will be translated to their corresponding VEP machine language representation. The translation is done in accordance with the size of operations and the adapted size of operands.

4.2 Compiler

Since writing a proof generator program in a high-level language is more convenient than writing it in the VEP assembly language, we implemented a C compiler which generates the assembly code.

At the first step, we generated a scanner for C language using the tool JFLex[1] which produces Java language source code. The lexical specification that we used was originally on the ones Jeff Lee published in 1985. Then we used the tool CUP[2], to write a parser for C language. Using CUP involves creating a simple specification based on the grammar for which a parser is needed. The grammar that we used was based on the ANSI C grammar also published by Jeff Lee though minor changes have been made.

The C compiler that we intended to implement was simpler than a complete C compiler because we wanted to use the compiler to compile a specific application written in a subset of C language. Thus, we used a free yet incomplete C compiler which could only print out a C code using the abstract syntax tree (AST) it has made with the help of the scanner and parser that we generated earlier. In this way, we had to implement two big and important phases of type checking and the code generation to a working compiler. Next, we explain these two phases.

4.2.1 Type checking phase

Each programming language has a type system which defines the way in which it classifies values and expressions into types. In our compiler (ANSI C language), we have the following types.

- *Arithmetic type*: Integral type (`int`,`char`) and Floating point type (`float`,`double`). Up to five sizes of integral types (`signed` and `unsigned`) are available: `char`, `short`, `int`, `long`, and `long long`. Up to three sizes of floating point types are available.
- *Derived Types*: arrays of objects of most types, functions that return objects of a

[1] JFlex is a lexical analyzer generator (also known as scanner generator) for Java, designed to work together with the LALR parser generator CUP. [50]

[2] Java Based Constructor of Useful Parsers (CUP for short) is a system for generating LALR parsers from simple specifications. CUP is written in Java, uses specifications including embedded Java code, and produces parsers which are implemented in Java. [51]

given type, pointers to objects of a given type, structures that contain a sequence of objects of various types, and unions capable of containing any one of several objects of various types, are the derived types.

- *`void` Type*: the `void` type specifies an empty set of values. It is used as the type returned by functions that generate no value. The `void` type never refers to an object and therefore, is not included in any reference to object types.

Since we needed the compiler to compile a specific program, we implemented the type checking phase only for the types which were used in the program. In this way, the type checking phase was not implemented for `union`s `enum`s and `struct`s. While traversing the AST, as the compiler enters a function body, a compound statement or generally a scope, it creates a symbol table. A symbol table is a data structure used by the compiler, to keep track of identifiers that occur in the source program and associate them with the information relating to theirs declaration or appearance in the source such as their type, scope level, location of definition, and if they are used or not. It should be mentioned that it is legal in C to have an identifier and a label with the same spelling. Therefore, we keep them in separate tables. These tables are kept as annotations on the relevant nodes of the AST. Therefore, the compiler adds entries to the relevant table, when it sees labels and identifier declarations, and looks up entries in the table when it sees label and identifier uses. At the end of the traversal, each instance of every identifier is annotated by a table entry that can be updated with information about that identifier. The declaration and each use of the same bounded name refer to the same table entry object. The compiler reports undeclared and multiply declared variables.

In order to perform the typing analysis, the compiler makes a single bottom-up pass over the AST, assuming that the tree is properly annotated with symbol table nodes on the scope constructs and table entries on the identifiers. The compiler adds a type annotation to each node that is an instance of expression, so that, at the end of the traversal, each node in every expression has a type annotation. During the same pass, the compiler determines types for expressions and computes values for constant expressions. The compiler can determine the type of an expression, inferring it according to the types of the identifiers (symbol table entries), constants and operators which constitute that expression, where the types of constants and operators is given by the language rules.

Type conversions play a major role in the type inference process. The conversions depend on the specific operator and the type of the operand or operands. Here, as an example, we explain the conversions on operands of arithmetical type. These conversions consist of implicit and explicit type conversions. Implicit type conversions can

occur when an assignment is made to an lvalue[1] that has a different type than the assigned value, or a function is provided an argument value that has a different type than the parameter. It also happens when the value specified in the return statement of a function has a different type from the defined return type for the function. In the following list the first two items define how implicit arithmetical type conversion is performed and the last one defines the explicit one.

- *Type promotion*: an integral promotion is the conversion of one integral type to another where the second type can hold all the possible values of the first type. If any operand is of type `long double`, the result type is `long double`. If any operand is of type `double`, the result type is `double`. If any operand is of type `float`, the result type is `float`. The integral promotions are performed on each operand as shown in Figure 4.6:

If there is an operand of type:	**The type of the result would be:**
`unsigned long long`	`unsigned long long`
`long long`	`long long`
`unsigned long`	`unsigned long`
`long`	`long`
`unsigned int`	`unsigned int`
`otherwise`	`int`

Figure 4.6: The integral promotion rules

- *Types compatibility*: the concept of compatible types is introduced by ANSI C to determine whether or not an implicit conversion is permissible. After promotion, using the appropriate set of promotion rules, two non-pointer types are compatible if they have the same size, signed-ness, and integer or float characteristic, or, in the case of aggregates, are of the same structure or union type.

- *Casting*: The cast operator is used for explicit type conversions. This operator has the following form, where `T` is a type, and `expr` is an expression: `( T ) expr` It converts the value of `expr` to the type `T`.

During this phase, the compiler looks for type mismatches and makes other static semantic checks required by ANSI C, and reports any errors to the user.

[1] An expression which can appear as the destination of an assignment operator indicating where a value should be stored. for example, a variable or an array element are lvalues but the constant 42 and the expression i+1 are not.

4.2.2 Code generation phase

If the program is correctly typed, finally, the generation of the VEP assembly language is carried out. In the code generation phase, the compiler converts the AST constructed during the previous phases into a VEP assembler code. Our compiler does not generate an intermediate representation such as three address code[1], since it would involve an extra translation step that is unnecessary given that the compiler will not be performing any optimizations. There are several different issues to be addressed during code generation, including memory locations selection for each variable, generation of executable code to correspond to the control structures, expressions, and operations in the input program.

The code generator traverses each node on the AST and generates an equivalent assembly language statement. For a `label` statement, for example, the code generator generates a unique label. For an `if` statement, it generates a compare statement and then the appropriate jump statement with the labels as jump destinations for the `then` and `else` statements. The compiler needs to keep the addresses of the variables needed in the instructions working with variables. The addresses are integer numbers starting from 0. Each variable declaration increments it in accordance to all-in-1-word storage policy (i.e., our C compiler allocates a C variable of any basic types (`char`, `short`, etc.) to a VEP word).

```
1  void main()
2  {
3    int previous;
4    int result;
5    int sum;
6    unsigned int i;
7    unsigned int n;
8    n = 13;                    // we want the 13th Fibonacci number
9    previous = -1;
10   result = 1;
11   for (i = 0; i <= n; ++i)  // calculates result 0 1 1 2 3 5 8 …
12       {
13        sum = result + previous;
14        previous = result;
15        result = sum;
16       }
17   putc(result);              // outputs the result
18 }
```

Figure 4.7: A sample C code which calculates the 13th Fibonacci number

Figure 4.7 shows a source code which outputs the character representation of the

[1] A multiple-address form of representing intermediate code which includes three addresses, usually two addresses from which data are taken and one address where the result is stored.

13th Fibonacci number (the result is 233, so the output would be 'é' on the VEP[1]). Figure 4.8 shows the generated assembly code for the given Fibonacci C source code. In the generated assembly code in this figure, the instructions corresponding to each line of the c source code are marked by that line number in italic style.

```
1  load 0                              1
2  load 0                              3
3  load 0                              4
4  load 0                              5
5  load 0                              6
6  load 0                              7
7  load 5                              8
8  load 13                             8
9  peeki 0-2                           8
10 poke                                8
11 peek                                8
12 pop                                 8
13 load 1                              9
14 load 1                              9
15 load 0-1                            9
16 mul                                 9
17 peeki 0-2                           9
18 poke                                9
19 peek                                9
20 pop                                 9
21 load 2                              10
22 load 1                              10
23 peeki 0-2                           10
24 poke                                10
25 peek                                10
26 pop                                 10
27 load 4                              11
28 load 0                              11
29 peeki 0-2                           11
30 poke                                11
31 peek                                11
32 pop                                 11
33 LoopBegLBL1:
34 load LoopEndLBL1-LoopStmLBL1        11
35 load 4                              11
36 peek                                11
37 load 5                              11
38 peek                                11
39 leq                                 11
40 LoopStmLBL1:
41 jmprf                               11
42 load 3                              13
43 load 2                              13
44 peek                                13
45 load 1                              13
46 peek                                13
47 add                                 13
48 peeki 0-2                           13
49 poke                                13
50 peek                                13
51 pop                                 13
52 load 1                              14
53 load 2                              14
54 peek                                14
55 peeki 0-2                           14
56 poke                                14
57 peek                                14
58 pop                                 14
59 load 2                              15
60 load 3                              15
61 peek                                15
62 peeki 0-2                           15
63 poke                                15
64 peek                                15
65 pop                                 15
66 load 4                              11
67 peeki 0-1                           11
68 peek                                11
69 load 1                              11
70 add                                 11
71 peeki 0-2                           11
72 poke                                11
73 peek                                11
74 pop                                 11
75 load LoopBegLBL1                    11
76 jump                                11
77 LoopEndLBL1:
78 load 2                              17
79 peek                                17
80 output                              17
81 load 0                              17
82 pop                                 17
83 halt                                18
84
```

Figure 4.8: Generated assembly code for the C code presented in Figure 4.7

[1]The VEP uses Microsoft Windows-1252 character encoding of the Latin alphabet, used by default in the legacy components of Microsoft Windows in English and some other Western languages.

4.3 Proof Generator

The safety proofs in PCC are represented in Edinburgh Logical Framework (LF) [36]. The typical LF representation of the proofs are large, due to a significant amount of redundancy. The fact that proofs contain many repeated patterns of proof rules and redundant arguments, makes them suitable for data compression. Compressing the proofs can alleviate the problem of proof size in communications, because it enables devices to transmit or store the same amount of data in fewer bits. A compressed proof, can get uncompressed using the decompressor which corresponds to the compressing algorithm. Therefore, a bundle of the compressed proof and a VEP machine executable decompressor which can decompress the compressed proof can make a sample proof generator. Figure 4.9 shows the implemented components that work together to make the proof generator. As it is shown in this figure, in our experiment, we used our compiler, our assembler and Gzip, an off the shelf compressor. Gzip [54] is a popular data compression program that is based on the `DEFLATE` [55] algorithm, which is a combination of Huffman coding [57] and LZ77 [56].

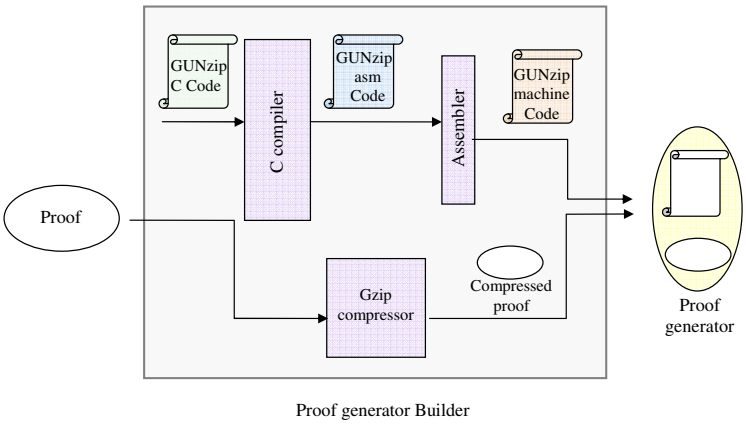

Figure 4.9: The components of the Proof generator Builder

In Figure 4.9, "GUNzip C code" is the Gzip source code stripped-down, so that, it only performs the `INFLATE` task (i.e., the decompression task which is the opposite of `DEFLATE`). That is, we extracted the decompressor part of the Gzip program. In this process, the extracted source code is modified in a way that it uses static allocation (which is done by a compilation switch). In order to facilitate the implementation, all the preprocessor commands and function prototypes are removed. Thus, we in-lined all the function. In order to in-line the functions without causing duplication of code and to avoid the code size increment, we used computed `goto`[1]. Since the computed `goto` is not supported by the ANSI C89 grammar, we added it to the grammar.

[1]A computed goto is a `goto` statement for which the address of the target is computed by an expression of type `void*`. It is possible to obtain the address of a label by applying the unary label

The reduced decompressor fetches its input (compressed data) from a literal string (array of compressed data) and outputs the decompressed data on the standard output. For the decompressor to fetch its input from a literal string, and to print a character, respectively, `readcmp` and `putchar` were developed as two special functions. These custom functions, get implemented using the VEP assembly language by the compiler as a runtime functionality.

The GUNzip C code is given to the compiler to generate the VEP assembly code of the GUNzip. This assembly code is then given to the assembler as input which results in having the GUNzip machine code as its output. Meanwhile, the proof is compressed by the Gzip compressor. Finally, the GUNzip machine code and the compressed proof are packed together as a proof generator. The packing is done manually by allocating the compressed stream statically in the code space which saves us a lot on stack space in comparison with the case dynamic allocation in a global variable. The compressed stream is then read by the proof generator using the special function `readcmp`.

It should be mentioned that among the mentioned components, the assembler can be regarded as a generic tool which can be used regardless of the preceding components which somewhat depend on the chosen technique of building the proof generator. In our experiment, the proof generator machine code without the compressed data is 10KB and the proof generator bundled together with the compressed proofs average 5% the original proofs which is about 20 times smaller than before.

4.4 Overall view

A detailed diagram of the implemented framework is presented in Figure 4.10. The producer is on the left-hand side and the consumer on the right-hand side of the figure. The producer builds a proof generator in the VEP machine language using the proof generator maker components. Before sending the proof generator, the producer has to add the request in code size, heap size, stack size, and execution time to the proof generator program header. For this, he has the option of running the proof generator on a copy of the VEP on his side. This custom-made copy of the VEP automatically adds the actual amount of the consumed resources to the proof generator program header, when the execution is finished.

---

value operator `&&` to the label. The target of a computed goto is known at run time. The language feature is an orthogonal extension to C99 and C++, implemented to facilitate porting programs, and developed with GNU C. [42]

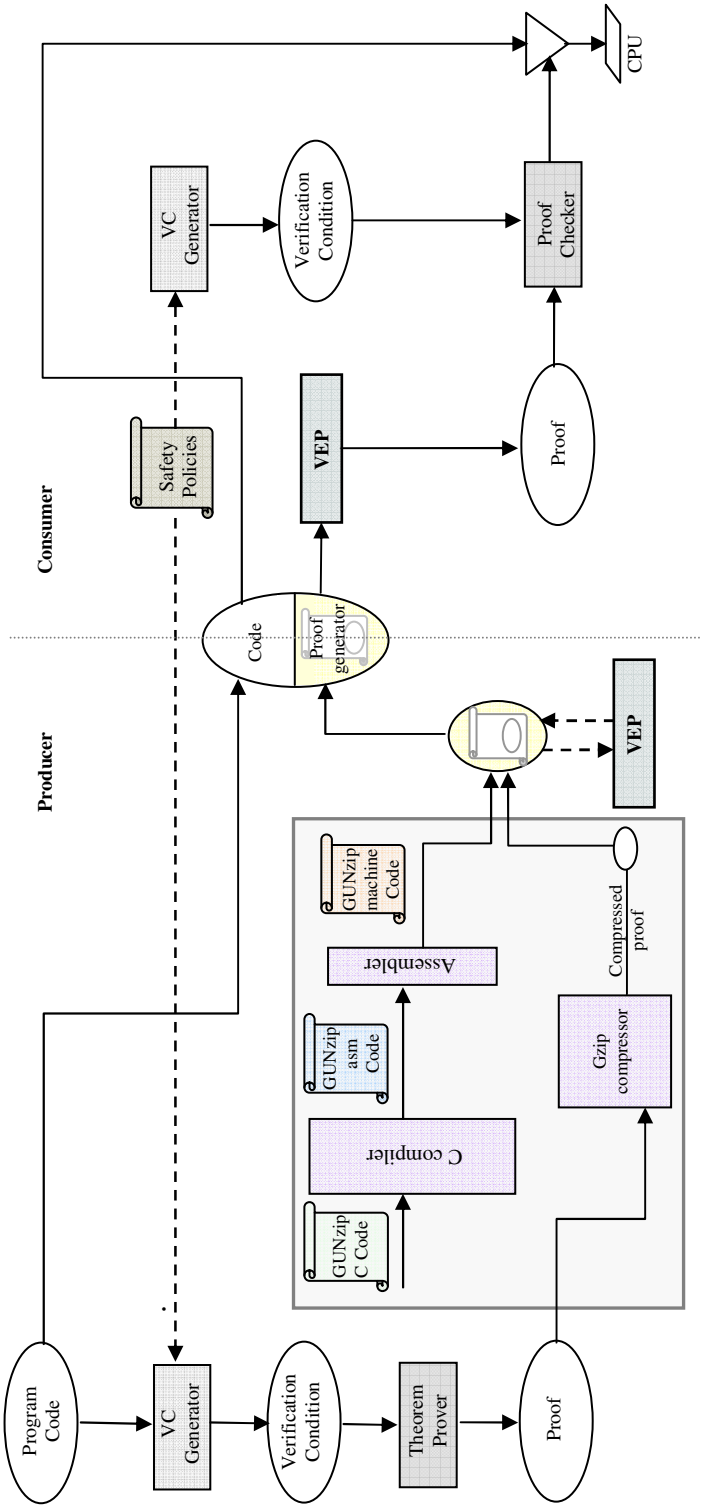

Figure 4.10: Detailed diagram of our sample implementation of EPCC

Chapter 5

Conclusion

The presence of untrusted and malicious codes and the absence of the necessary security bases and frameworks for safe use of the mobile codes are among the main concerns of our computer-dependent world. The problem with the current approaches is either because of their disobedience of the security principles or due to their hard applicability.

The trustworthiness of the PCC is an important advantage over approaches that involve the use of complex security systems on the consumer side. One of the key issues with the practical applicability of PCC and its related methods is the difficulty in communicating and storing the proofs which makes the approach less scalable. The approaches proposed to alleviate this suffer with drawbacks of their own, especially the enlargement of the trusted computing base, in which any bug may compromise the safety of the whole system.

We have worked on a hybrid approach with some modifications on the level of the framework which offers a solution to the PCC's scalability issue. In our approach, instead of transmitting the proofs, a proof generator for the code in question is sent. The new modified extended framework enables the execution of the proof generator on the consumer side in a secure manner. We showed the design of a generic extended framework for proof-carrying code (EPCC). The EPCC framework makes the PCC idea more scalable and practical while respecting the characteristics of the PCC technique. EPCC provides the code consumer with the luxury of using a safe environment in which a big class of proof generators can be executed in a secure manner, regardless of the original logic in which the proofs were represented. We presented the design of a safe and small Virtual machine (VEP), which is introduced in the EPCC framework. For this, we implemented the VEP to work as an online reference monitor. We showed empirically that the EPCC and its conjoint virtual machine VEP have the potential

to be used in an industrial-strength framework. To show this, we implemented the necessary programs to complete the end-to-end chain. For this, we implemented an assembler for the VEP byte code. We also implemented a C compiler to target the assembler language. Finally, in order to have a runnable EPCC framework, we made use of GUnzip to build a sample proof generator. In this way, EPCC leaves the easier tasks to the consumer (like PCC) and gives adequate means to the producer to do the hard task (to the contrary of PCC). This major flexibility for the consumer and producer, in addition to the alleviation of the proof size issue, are gained through a minor TCB extension of less than 300 lines of code which can be verified easily by pen and paper. In this way, EPCC intends to have a small TCB and give the highest priority to the security.

5.1 Future work

In the future, there are several directions to extend the approach that we proposed in this thesis.

Clearly, a first practical step will be to apply the framework to an other possible PCC techniques like FPCC and OPCC. Our sample implemented of the EPCC framework can be used to make the FPCC approach more scalable. As for the OPCC, writing an oracle-based proof generator could be a possible direction to explore. This proof generator could be one which uses the oracle in order to rebuild the original proof. Therefore, there would be no need to use any non-deterministic proof checker on the consumer side and the verification could be done with the original PCC proof checker. In this way, we will not force the consumer to change the PCC structure to gain the benefit of small proofs from using OPCC and there will be no need for compromises in the size of TCB.

One other direction to extend our approach is writing a proof generator which can translate a proof from a higher-order logic [58], into a similar proof written in the logic set by the consumer. In other words, the producer has the possibility of reducing the size of the safety proof by writing it in a custom logic which can be later translated to the logic set by the consumer by the proof generator.

Appendix A

Instruction Set Descriptions

A.1 Stack Manipulation Instructions

A.1.1 POP

- *Opcode:*
  `00101000`
- *Corresponding Assembly Instruction*
  `pop`
- *Description:*
  Removes the an item from the top of the stack.

A.1.2 POKE

- *Opcode:*
  `00000001`
- *Corresponding Assembly Instruction:*
  `poke`
- *Description:*

Pops[1] p, pops q, then writes q to the stack location specified by p. If p is positive, the writing destination is the stack position p from the bottom of the stack. Otherwise, it is the stack position SP+p, where SP is the stack pointer.

A.1.3 PEEK

- *Opcode:*

  `00000000`

- *Corresponding Assembly Instruction:*

  `peek`

- *Description:*

  Pops p, if p is positive, the content of the stack position p from the bottom of the stack is pushed onto the stack. Otherwise, the content of the stack position SP+p is pushed onto the stack, where SP is the stack pointer.

A.1.4 LOAD1

- *Opcode:*

  `00011100 B1`

- *Corresponding Assembly Instruction:*

  `load` *expr*

- *Description:*

  Pushes the numeric value encoded by the next byte in the code space onto the stack.

A.1.5 LOAD2

- *Opcode:*

  `00011101 B1 B2`

[1] The "push" and "pop" used to explain the instructions are not done incrementally and the stack pointer is set at the end of the execution of the instruction (e.g., the POKE instruction reads the first and the second element of the stack from the top and writes the second element to the stack location specified by the first element then does two consecutive pops).

- *Corresponding Assembly Instruction:*

  `load` *expr*

- *Description:*

  Pushes the numeric value encoded by the next two byte in the code space onto the stack in big-endian format.[1]

A.1.6 LOAD3

- *Opcode:*

  `00011110 B1 B2 B3`

- *Corresponding Assembly Instruction:*

  `load` *expr*

- *Description:*

  Pushes the numeric value encoded by the next three byte in the code space onto the stack in big-endian format.

A.1.7 LOAD4

- *Opcode:*

  `00011111 B1 B2 B3 B4`

- *Corresponding Assembly Instruction:*

  `load` *expr*

- *Description:*

  Pushes the numeric value encoded by the next four byte in the code space onto the stack in big-endian format.

A.1.8 PUSH-PC

- *Opcode:*

  `00000011`

[1] A format for storage of binary data in which the most significant bit is placed first.

- *Corresponding Assembly Instruction:*

  `push-pc`

- *Description:*

  Pushes the current value of program counter (which points to the PUSH-PC instruction) onto the stack.

A.1.9 READC

- *Opcode:*

  `00000100`

- *Corresponding Assembly Instruction:*

  `readc`

- *Description:*

  Pops p, then, pushes the byte found in code space at the position specified by p onto the stack.

A.2 Program Control Instructions

A.2.1 HALT

- *Opcode:*

  `00000110`

- *Corresponding Assembly Instruction:*

  `halt`

- *Description:*

  Is used to show the end of the program and stop the code execution.

A.2.2 NOP

- *Opcode:*

`00000010`

- *Corresponding Assembly Instruction:*

  `nop`

- *Description:*

  No operations are performed by NOP.

A.3 Arithmetical Instructions

A.3.1 ADD

- *Opcode:*

  `00000111`

- *Corresponding Assembly Instruction:*

  `add`

- *Description:*

  Pops p, pops q, adds q to p and pushes the result onto the stack.

A.3.2 SUB

- *Opcode:*

  `00001000`

- *Corresponding Assembly Instruction:*

  `sub`

- *Description:*

  Pops p, pops q, subtracts p from q (i.e. $q-p$) and pushes the result onto the stack.

A.3.3 MUL

- *Opcode:*

  `00001001`

- *Corresponding Assembly Instruction:*

  `mul`

- *Description:*

  Pops p, pops q, multiplies q by p and pushes the result onto the stack.

A.3.4 DIV

- *Opcode:*

  `00001010`

- *Corresponding Assembly Instruction:*

  `div`

- *Description:*

  Pops p, pops q, divides q into p and pushes the result onto the stack.

A.3.5 MOD

- *Opcode:*

  `00001011`

- *Corresponding Assembly Instruction:*

  `mod`

- *Description:*

  Pops p, pops q, calculates the remainder, on division of q by p and pushes it onto the stack.

A.4 Comparison Instructions

A.4.1 EQU

- *Opcode:*

  `00001100`

- *Corresponding Assembly Instruction:*

  `equ`

- *Description:*

  Pops p, pops q, if q and p have the same value and type, 1 is pushed onto the stack. Otherwise, 0 is pushed onto the stack.

A.4.2 NEQ

- *Opcode:*

  `00001111`

- *Corresponding Assembly Instruction:*

  `neq`

- *Description:*

  Pops p, pops q, if q and p have the same value and type, 0 is pushed onto the stack. Otherwise, 1 is pushed onto the stack.

A.4.3 LTH

- *Opcode:*

  `00001101`

- *Corresponding Assembly Instruction:*

  `lth`

- *Description:*

  Pops p, pops q, if q is less than p, 1 is pushed onto the stack. Otherwise, 0 is pushed onto the stack.

A.4.4 LEQ

- *Opcode:*

  `00001110`

- *Corresponding Assembly Instruction:*

  `leq`

- *Description:*

  Pops p, pops q, if q is than or equal to p, 1 is pushed onto the stack. Otherwise, 0 is pushed onto the stack.

A.4.5 ISPAIR

- *Opcode:*

  `00010111`

- *Corresponding Assembly Instruction:*

  `ispair`

- *Description:*

  Pops p, if p has the pair type, pushes a 1 onto stack. Otherwise, pushes a 0 onto stack.

A.5 Bitwise Logical Instructions

A.5.1 BAND

- *Opcode:*

  `00010000`

- *Corresponding Assembly Instruction:*

  `band`

- *Description:*

  Pops p, pops q, performs a bitwise AND operation on q and p (for each bit, if both bits are 1, the resulting bit is a 1, otherwise it is a 0), then pushes the result onto the stack.

A.5.2 BOR

- *Opcode:*

  `00010011`

- *Corresponding Assembly Instruction:*

  `bor`

- *Description:*

  Pops p, pops q, performs a bitwise OR operation on q and p (for each bit, if both bits are 0, the resulting bit is a 0, otherwise it is a 1), then pushes the result onto the stack.

A.5.3 BNOT

- *Opcode:*

  `00010010`

- *Corresponding Assembly Instruction:*

  `bnot`

- *Description:*

  Pops p, performs a bitwise NOT operation on p (for each bit in the output value, if the input bit is a 0, the output bit is set to a 1. Otherwise it is set to a 0), then pushes the result onto the stack.

A.5.4 BSHIFT

- *Opcode:*

  `00010001`

- *Corresponding Assembly Instruction:*

  `bshift`

- *Description:*

  Pops p, pops q, if p is positive, shifts q to the left by the number of places specified by p. Otherwise, shifts q to the right by the number of places specified by p. The

sign of q is preserved (so a negative number will have ones added to the left, while a positive number will have zeroes added to the left).

A.6 Heap Manipulation Instructions

A.6.1 CONS

- *Opcode:*

  `00010100`

- *Corresponding Assembly Instruction:*

  `cons`

- *Description:*

  Pops p, pops q, then constructs a new pair containing (q, p), in the heap, and pushes the heap address of the built pair onto the stack.

A.6.2 CAR

- *Opcode:*

  `00010101`

- *Corresponding Assembly Instruction:*

  `car`

- *Description:*

  Pops p, then pushes the left child of the pair with the heap-address specified by p onto the stack.

A.6.3 CDR

- *Opcode:*

  `00010110`

- *Corresponding Assembly Instruction:*

  `cdr`

- *Description:*

  Pops p, then pushes the right child of the pair with the heap-address specified by p onto the stack.

A.7 Jump Instructions

A.7.1 JUMP

- *Opcode:*

  `00011000`

- *Corresponding Assembly Instruction:*

  `jump`

- *Description:*

  Pops p, sets the program counter (PC) to the address p in code space and continues execution from that address.

A.7.2 JMPR

- *Opcode:*

  `00011001`

- *Corresponding Assembly Instruction:*

  `jmpr`

- *Description:*

  Pops p, sets the program counter (PC) to the address PC+p in code space (p is an offset which can be negative or positive), continues execution from that address.

A.7.3 JMPRT

- *Opcode:*

  `00011010`

- *Corresponding Assembly Instruction:*

  `jmprt`

- *Description:*

  Pops p, pops q, if the value p is 0, the execution is continued as normal with the instruction following the JMPRT. Otherwise, sets the program counter (PC) to the address PC+q in code space (q is an offset which can be negative or positive), continues execution from that address.

A.7.4 JMPRF

- *Opcode:*

  `00011011`

- *Corresponding Assembly Instruction:*

  `jmprf`

- *Description:*

  Pops p, pops q, if the value p is 0, sets the program counter (PC) to the address PC+p in code space (p is an offset which can be negative or positive), continues execution from that address. Otherwise, the execution is continued as normal with the instruction following the JMPRF.

A.8 Compact Instructions

A.8.1 POKE-*i*

- *Opcodes:*

  `00101001 - 00101111`

- *Corresponding Assembly Instruction:*

  `pokei` *expr*

- *Description:*

  Pops p, writes p in the stack position SP-i, where SP is the stack pointer and i is specified by the lower four bits of the opcode.

A.8.2 PEEK-i

- *Opcode:*

  `0010xxxx`

- *Corresponding Assembly Instruction:*

  `peeki` *expr*

- *Description:*

  Pushes the content of the stack position SP-i onto the stack, where SP is the stack pointer and i is specified by the lower four bits of the opcode.

A.8.3 LOADi

- *Opcode:*

  `1xxxxxxx, 01xxxxxx, 0011xxxx`

- *Corresponding Assembly Instruction:*

  `load` *expr*

- *Description:*

  Pushes the value i onto the stack, where the value i depend on the opcode. If the opcode is of the form `1xxxxxxx` then i is the value specified by the lower 7 bits. If the opcode is of the form `01xxxxxx` then i is equal to the subtraction of the value specified by the lower 6 bits from -1. Otherwise, i is equal to the subtraction of the value specified by the lower 4 bits from -65.

Bibliography

[1] The Mars Rover Spirit FLASH anomaly. Aerospace Conference, 2005 IEEE Volume, March 2005 Pages: 4186 - 4199.

[2] T. R. Weiss. Out-of-memory problem caused Mars rover's glitch. Computerworld. http://www.computerworld.com, February 4 2004.

[3] CA Inc. Internet Security Outlook Report. January 2008.

[4] R. Grimm , B. Bershad, Security for Extensible Systems, Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), p.62, May 05-06.

[5] A Language-Based Approach to Security, Fred B.Schneider, Greg Morrisett, and Robert Harper. Cornell University, Ithaca, NY Carnegie Mellon University, Pittsburgh.

[6] D. Kozen. Language-based security. In Mathematical Foundations of Computer Science, pages 284-298, 1999

[7] Saltzer, J.H. And Schroeder, M.D., The Protection of information in computer systems, Proceedings of the IEEE, vol. 63, no. 9 (Sept 1975), pp. 1278-1308.

[8] The ISOS Years: Madrid 1941-3 Kenneth Benton Journal of Contemporary History, Vol. 30, No. 3 (Jul., 1995), pp. 359-410

[9] Department of Defense: Trusted Computer System Evaluation Criteria, 1983, http://ftp.std.com/obi/DOD/orange.book/

[10] Microsoft Corporation, Proposal for authenticating code via the Internet, April 1996

[11] Easter eggs in Microsoft products, http://en.wikipedia.org/wiki/Easter_eggs_in_Microsoft_products

[12] U. Erlingsson and F.B. Schneider, "SASI Enforcement of Security Policies: A Retrospective," Proc. New Security Paradigms Workshop (NSPW 99), ACM Press, 1999, pp. 87-95.

[13] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In Proceedings of the 1999 IEEE Symposium on Security and Privacy, pages 32-45, Oakland, California, May 1999

[14] Lujo Bauer, Jay Ligatti, and David Walker, 2005. Composing security policies with polymer. In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 305-314. ACM Press, New York, NY, USA. ISBN 1-59593-056-6.

[15] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In Xavier Leroy and Atsushi Ohori, editors, Proc. Workshop on Types in Compilation, volume 1473 of Lecture Notes in Computer Science, pages 28-52. Springer-Verlag, March 1998.

[16] 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 Proc. Workshop on Compiler Support for System Software, pages 25-35. ACM SIGPLAN, May 1999

[17] George C. Necula. Proof-carrying code. In Proc. 24th Symp. Principles of Programming Languages, pages 106-119. ACM SIGPLAN/SIGACT, January 1997.

[18] George Necula & Peter Lee, Proof-Carrying Code, Technical Report CMU-CS-96-165, 1996.

[19] George Necula. Proof-carrying code. In 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, pages 106-119, New York, January 1997. ACM Press.

[20] George Ciprian Necula, Compiling with Proofs, CMU-CS-98-154, September 1998.

[21] G. Necula, Proof-Carrying Code. In Benjamin C. Pierce, ed., Advanced Topics in Types and Programming Languages, MIT Press, 2005.

[22] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In 6th ACM Conference on Computer and Communications Security. ACM Press, November 1999.

[23] G. Necula and P. Lee, Safe Kernel Extensions Without Run-Time Checking, in Proceedings of the 2nd Symposium on Operating System Design and Implementation (OSDI'96), Seattle, October, 1996, 229-243.

[24] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In Proceedings of the Conference on PLDI'00, Vancouver, Canada, June 2000, 95-107.

[25] P. Lee and G. Necula, "Research on Proof-Carrying Code on Mobile-Code Security" In Proceedings of the Workshop on Foundations of Mobile Code Security, 1997

[26] George C. Necula: A Scalable Architecture for Proof-Carrying Code. FLOPS 2001: 21-39.

[27] George C. Necula, Shree Prakash Rahul: Oracle-based checking of untrusted software. POPL 2001: 142-154.

[28] Appel, A. W. (2001). Foundational proof-carrying code. In 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01), pages 247-258.

[29] Dinghao Wu, Andrew W. Appel, Aaron Stump, Foundational Proof Checkers with Small Witnesses, Uppsala, Sweden, 27-29 August 2003.

[30] Christopher League, Zong Shao, and Valery Trifonov. Precision in practice: A type-preserving java compiler. Technical report, YALE DCS TR-1223, January 2002.

[31] Dinghao Wu, Andrew W. Appel, Aaron Stump: Foundational proof checkers with small witnesses. PPDP 2003: 264-274.

[32] Shree P. Rahul and George C. Necula. "Proof Optimization Using Lemma Extraction". UCB Technical Report CSD-01-1143, May 2001.

[33] First-Order Term Compression: Techniques And Applications, James R. Cheney, Carnegie Mellon University, 1998.

[34] Th Computational Beauty of Nature, Computer Explorations of Fractals, Chaos, Gary William Flake, The MIT Press; 1 edition, July 10, 1998. Complex Systems, and Adaptation

[35] Zhong Shao. An overview of the FLINT/ML compiler. In ACM SIGPLAN Workshop on Types in Compilation (TIC) , Amsterdam, The Netherlands, June 1997.

[36] Harper, R., F. Honsell and G. Plotkin, A Framework for Defining Logics, Journal of the Association of Computing Machinery 40 (1993), pp. 143Ű184.

[37] Joyce, E., Is Error-Free Software Achievable Datamation, Feb.18, 1989.

[38] Fenton, N., and Neil, M., A critique of software prediction models. IEEE Trans. On Software Engineering, Vol., 25, No.5, 1999.

[39] Robert V. Binder, "Six Sigma: Hardware Si, Software No!", http://www.rbsc.com/pages/sixsig.html, 1997.

[40] Computer Architecture: A Quantitative Approach, Third Edition

[41] IBM Terminology, IBM Corp, March 2004.

[42] The information center for IBM's Linux compilers, March 15, 2005.

[43] GUNZip, www.gzip.org/

[44] R. Ierusalimschy, L.H. de Figueiredo, and W. Celes. The implementation of Lua 5.0. Journal of Universal Computer Science, 11(7):1159-1176, 2005.

[45] Neil C. Brown. Rain VM: Portable Concurrency through Managing Code. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, Communicating Process Architectures 2006, pages 253-267, September 2006.

[46] Klaas-Jan Stol, Architecture of the Parrot virtual machine, 17.10.2005

[47] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. The Java Series. Addison Wesley Longman, Inc., second edition, April 1999.

[48] Meijer Erik, Gough, John: Technical Overview of the Common Language Runtime. Microsoft Research, 2002. http://research.microsoft.com/~emeijer/

[49] Appel, A. W. and Wang, D. C.: JVM TCB: Measurements of the trusted computing base of Java virtual machines, Technical Report CS-TR-647-02, Princeton University, 2002.

[50] JFlex - The Fast Scanner Generator for Java, http://jflex.de/

[51] CUP - LALR parser generator for Java, http://www2.cs.tum.edu/projects/cup/

[52] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In Interpreters, Virtual Machines and Emulators (IVME 03), pages 41-49, 2003.

[53] Alfred V. Aho, Ravi Sethi, and Jerey D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1988.

[54] P. Deutsch, GZIP file format specification version 4.3, May 1996, http://www.ietf.org/rfc/rfc1952.txt.

[55] P.Deutsch, DEFLATE Compressed Data Format Specification version 1.3, May 1996, http://www.ietf.org/rfc/rfc1951.txt.

[56] J. Ziv and A. Lempel, A universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, vol. IT-23, May 1977.

[57] D. Huffman. A method for the construction of minimum redundancy codes. Proceeding. of IRE 40:1098-1101, 1952.

[58] Miller, Dale, "Logic: Higher-order," Encyclopedia of Artificial Intelligence, 2nd ed, 1991.