Encoding the program correctness proofs as programs in PCC technology*

Heidar Pirzadeh
DIRO, University of Montreal
CP 6128 succ Centre-Ville
Montreal (QC), Canada, H3C 3J7
Pirzades@iro.umontreal.ca

Danny Dubé
Département d'informatique et génie logiciel
Université Laval
Québec (QC), Canada, G1V 0A6
Danny.Dube@ift.ulaval.ca

Abstract

One of the key issues with the practical applicability of Proof-Carrying Code (PCC) and its related methods is the difficulty in communicating and storing the proofs which are inherently large.
The approaches proposed to alleviate this, suffer from 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 a generic extended PCC framework (EPCC) in which, instead of the proof, a proof generator for the program in question is transmitted. This framework enables the execution of the proof generator and the recovery of the proof on the consumer side in a secure manner.

1. Introduction

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 the one hand and lack of security due to its expense to set up and annoyance to get along with, on the other hand, makes it easier for intruders to implement their plans.

Malicious software not only grows in quantity (16 times in year 2007) but also becomes more sophisticated [5]. 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 [30]. 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 [14]. The explorations on the domain of programming languages to find techniques that can help us enforce the necessary security policies to computing systems are called language-based security approaches [32]. Language-based security conceptually is a combination of two classic computer security principles:

1. *Least privilege*: throughout execution, each application should be given the minimum resources necessary to accomplish its task [31].

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.

The remainder of this paper is organized as follows. In Section 2, we introduce the existing methods and motivate the need for a framework that can overcome the obstacles of the current ones. For this, we take a glimpse at related work and assess the merits and drawbacks of each approach, from our perspective. In Section 3, we present and describe our generic Extended Proof-Carrying Code (EPCC) framework. Sections 4 to 8 discuss the design of the virtual machine for EPCC (the VEP) and the way in which the VEP works. Having all these factors set, in Section 9, we make the whole system work, bridging from theory to practice, by presenting a sample use of an EPCC framework through an implemented prototype. Finally, we summarize and give an outlook of future work in Section 10 and present our conclusions in Section 11.

2. Existing Methods

2.1. Classes of Approaches

Authentication is a class of approaches in which one accepts only the codes that come equipped with the signature of a trusted producer. Ideally, since this producer is trusted,

*This work was supported by the Natural Sciences and Engineering Research Council of Canada

the code can be also trusted. This class of approaches has several drawbacks. Applying this method on a large scale, as a solution to the untrusted codes problem, is against the second principle of security design. The principle of Minimum TCB applies to any trust management system, where the trusted party is the TCB and the trusting party is the computing system. Considering the trusted companies to be part of the TCB leads to a considerable enlargement of the TCB. Furthermore, trusting a producer does not necessarily mean that the produced code is safe and performs as claimed.

Dynamic code analysis techniques are the techniques that observe the execution of a code and perform an appropriate action before the code violates the security policy. 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* (e.g., operating system, virtual machine, wrapper program, etc.). 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 from some drawbacks. The monitor has to work every time we run the code and 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. This stoppage 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. An 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, the safety of the code cannot be proven through execution monitoring. The drawbacks of the execution monitors are the consequences of execution-time security enforcement.

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 target computing system (i.e. the code consumer) with a safety verifier which can prove the safety of every untrusted code upon receiving them. Since verifying the safety of a received code is a hard task, the safety verifier would, inevitably, be a complex program. "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.

2.2. Proof-Carrying Code

The main idea behind the Proof-Carrying Code (PCC) approach [22, 21, 13, 24] is to shift a large part of the responsibility from the code consumer to the code producer. This is done by breaking the safety verifier 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 code consumer side (i.e. in the TCB). In this way, the burden of proving the safety of the untrusted code is put on the shoulders of the producer.

The PCC enables the consumer to verify that a piece of code it has received from the untrusted producer complies with its safety policy. The safety policy is specified by means of a set of axioms and rules that the code producer can use for the purpose of constructing a proof. Using the verification condition generator (VCGen), the consumer constructs a verification condition (VC) which is a formula in a certain logic. The VC has the property that it is provable only if the code respects the safety policy. The constructed VC then is sent to the code producer (or the producer, given a copy of the VCGen, can construct the VC himself). The code is accompanied by what the code producer claims to be the proof of the VC. Before executing the code, the consumer uses a proof checker to verify that the received proof is indeed a proof of the VC. If so, the code is safe and can be executed. Figure 1 shows the interaction between the entities involved.

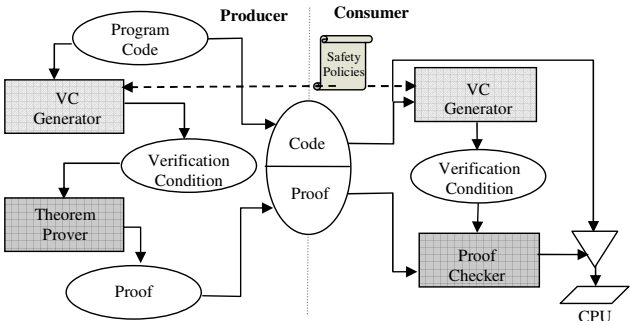

Figure 1. Traditional PCC framework

Since PCC is based on static analysis, once the safety of an untrusted code is successfully established, there is no need to check the code anymore. As a result, we have a computing system with less overhead and more security [2, 25]. The following fundamental characteristics of the PCC approach nominate it as one of the strongest frameworks to be used in mobile-code security:

(1) PCC gives the highest priority to the security (*raison d'être*);

(2) it intends to have a small TCB;

(3) it leaves the easier tasks to the consumer;

(4) while it is tamperproof.

2.3. PCC Variants

The traditional PCC 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 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 the proofs because they are possibly sent through communication networks. This difficulty of communicating the proofs, which are inherently large, makes the PCC less scalable. In traditional PCC framework, it is not unusual to see proofs that are 1000 times larger than the associated code, which makes the use of PCC impractical for all but the tiniest examples [26].

The size of the TCB in proof-carrying code is about 15000 to 20000 lines of code. Any bug in these components can compromise the security of the whole system. One can 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) [17, 12]. However, leading edge software development organizations typically achieve a defect density of about 2 defects/KLOC [4]. 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 cannot easily be trusted. That is, the anxiety 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.

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 finds 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.

Any solution to combat these obstacles and to make a refinement in the PCC technology has to respect the mentioned fundamental characteristics of the PCC approach. In order to reduce the TCB, Appel et al [1] introduced the notion of foundational proof-carrying code (FPCC).

Although the TCB components in the traditional PCC framework are simple, Appel pointed out that the VCGen (and consequently the TCB) is too large [1] and it needs to be verified itself. FPCC, in principle, is strictly more likely to be secure than traditional PCC because it has a smaller trusted computing base. As it is shown in Figure 2, in this

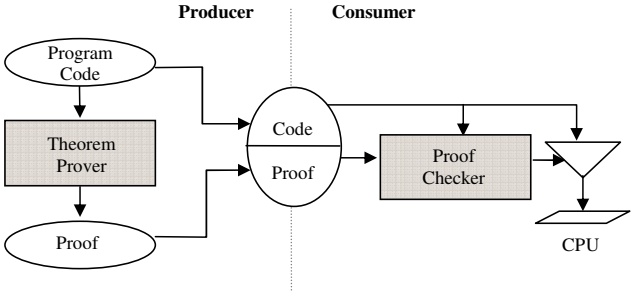

Figure 2. Foundational PCC framework

technique, VCGen is removed from the consumer side, and the TCB becomes minimal.

The proofs in FPCC, in comparison with traditional PCC, are more complicated to produce. The size of the proofs in FPCC is 20% bigger than that in traditional PCC and can explode exponentially. Therefore, the proof size, which is a crucial obstacle for the practical applicability of traditional PCC and related techniques, remains unsolved. This makes the proof communication harder and the use of FPCC even less practical than that of traditional PCC [26].

Regarding the proof-size obstacle in traditional PCC, Necula proposed a new strategy called Oracle-based Proof-Carrying Code (OPCC) [26]. In this approach, the handling of the proofs on the consumer side is changed. As shown in Figure 3, this change in strategy led to a change in the framework, namely, they assumed that the consumer uses a non-deterministic proof checker. An untrusted theorem prover on the left-hand side records a sequence of bits that indicates which sub-goals failed and needed backtracking. Then, the producer sends this bit stream to the consumer and serves as a proof witness. 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. The oracle string 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. 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.

One of the biggest downsides of the OPCC is that it involves complex trusted components, such as a non-deterministic proof checker plus the usual PCC components. The trusted computing base in OPCC is about 26000 lines of code which is bigger than the TCB size in traditional PCC. Any flaw in the implementation of these components can compromise safety of the system. 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

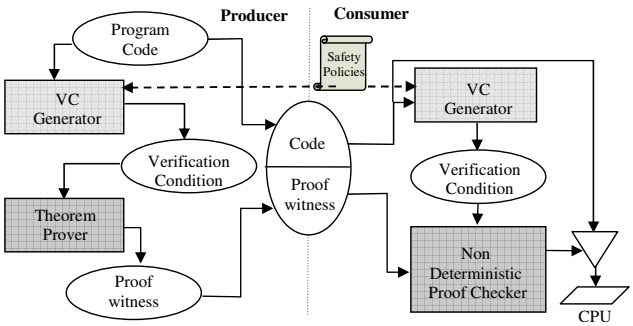

Figure 3. Oracle-based PCC framework

system [7] showed a critical leak in its type axioms found by League [18]. Unfortunately, the big size of the TCB in OPCC goes against the first and the second characteristics of the PCC approach.

In contrast to these developments, 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.

3. Extended Proof-Carrying Code Framework

As we mentioned earlier, one of the crucial issues for the practical applicability of PCC 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 rewritten in a more compact form which preserves the meaning of the proof of the original form [29, 6]. This could be done finding for a given term t a smaller equivalent term s and replacing all the occurrences of t with s in the proof (e.g., in the arithmetic system, there could be a rule $x + 0 \rightarrow x$ which always reduces the size of a term). Using proof optimization in an approach called lemma extraction, Necula *et al.* could not obtain a reduction better than 15% in the size of the proofs.

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 the 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 a decompressor would be a specific decompresser that cannot 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. In the OPCC approach, Necula *et al.* used the idea of the proof compression. Although their approach resulted in proofs which were smaller than the original proofs, they payed the price of a considerable enlargement of the TCB [26, 33, 23]. We are not in favor of compromising the security of the system with a big TCB expansion simply because the proofs are too large.

We present an extended framework that allows the PCC proofs to be represented as programs. This contributes to reduce the negative impact of the size of the proof and enables the PCC to handle even very large programs. The idea of representing the proofs as programs is inspired by the *Kolmogorov complexity*. Roughly speaking, the Kolmogorov complexity of a string x is the shortest computer program that produces x, 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 of the shortest program capable of producing x on a universal computer U (such as a Turing machine). Note that Kolmogorov 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 proposed 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*[1] 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.

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 4. In an EPCC system, there are two main parties, a code producer, who

[1] A proof generator is a program whose sole function is to output the proof. This program aims to be a more compact representation of its resulting proof and does not necessarily rediscover the proof.

sends a code along with its safety proof generator, on the left-hand side, and a code consumer, who wishes to run the code, provided that it is proven safe by the system, on the right-hand side.

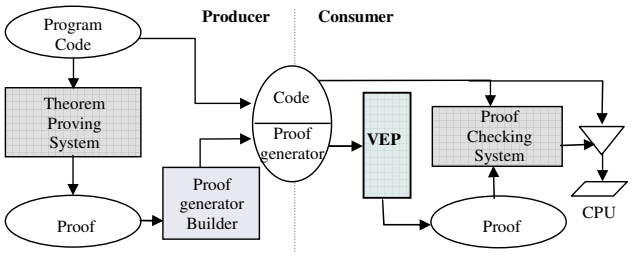

Figure 4. The generic EPCC framework

The communication between these two parties may consist of a multi-step interaction between the producer and the consumer depending on the underlying 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. 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 (the VEP) [28] 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 is tamper proof, like PCC.

One of the crucial components in the EPCC framework is the VEP which is a universal computer in the trusted computing base of the EPCC. The 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. In the following, we discuss the ways in which the VEP provides us with the necessary basis to apply the Kolmogorov complexity idea and enables the execution of the proof generator on the consumer side in a secure manner.

4. The VEP Virtual Machine

The most popular virtual machines, like Java Virtual Machine [19] and Common Language Runtime [20], use a stack machine type rather than the register-oriented architectures used in real processors, due to the simplicity of their implementation. 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.

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. [8] 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 the 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.

The VEP uses three blocks of memory: a code space (whose cells are bytes), a heap (made to contain pairs of machine words), and a stack (whose cells are machine words). The stack in the VEP is ascending (i.e. the stack grows towards the high addresses) and conforms to the full stack convention (i.e. the stack pointer points at the topmost element) and the first item pushed on the stack is stored at address zero. The heap provides the programmers with additional flexibility by supplying the VEP with memory for objects of arbitrary lifespan. The VEP also provides automatic memory management of the heap, thus there can 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.

The VEP provides 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. The current implementation of the VEP is a stack-based machine with random access to the stack which makes it Turing complete. The VEP executes 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" because the VEP provides us with random access to the code and the stack cells). Knowing that the VEP has finite resources, the

question pops up whether it can be considered as 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.

On the VEP, we have two distinct types of values: *numbers* and *pairs*. Considering that the VEP is implemented using 32-bits machine words, 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. If we have a cell that references a *pair*, the contents of the cell represents the address of a pair in the heap memory. For a cell with its type *number*, the contents of the cell is a signed integers.

The VEP has a RISC-like instruction set which provides random access to stack, plenty of arithmetic, logical, comparison, data transfer, and control instructions and restricted access to the pair-based heap. This gives application developers tremendous flexibility in implementing their ideas and innovations. The code space, being made of bytes, naturally leads to an instruction set of 256 instructions. Since we intend to execute proof generators on the VEP and they might execute more efficiently on a machine with rich set of data transfer instructions, we assign 224 of the opcodes to the data transfer instructions.

In the EPCC framework, the proof generators are untrusted programs which have to be executed on the consumer side. Since running untrusted programs on the consumer side is against the *raison d'être* 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 provides a tightly-controlled set of resources for proof generators to run in. In order to be able to output the resulting proof, a proof generator is allowed to print characters onto the standard output. This is the sole way provided by the VEP for a proof generator to communicate with the outside world. Other than that, network access, the ability to inspect the host system, or reading from input devices and writing into file streams are disallowed. In this sense, the VEP performs the dynamic analysis.

As we mentioned earlier, two main drawbacks of the dynamic analysis approaches are their high overhead and their fail-stop manner in case an unsafe code is encountered. Here, we discuss the existence of each of these issues. An unbounded *number of runs* and/or *execution time* of a virtual machine, each, incurs an unbounded cost on the system that uses the execution monitor. In the case of EPCC, we need the VEP to run only for a single time, in which the proof generator outputs the proof or fails. Now, given that we can run the monitor for a limited period of time we can avoid the high overhead of the VEP. For that reason, the VEP runs for a limited number of CPU cycles, which is checked during the execution of the proof generator. In this way the VEP can enforce fine-grained memory safety, control-flow safety, and type safety with an insignificant overhead.

As for the second drawback of the dynamic analysis approaches, the fail-stop manner is aligned with the safety requirement of the EPCC framework. That is, we want the VEP to act in a fail-stop manner to prevent an unsafe proof generator to continue its execution. Therefore, not only does the fail-stop manner has no dangerous consequence but it is required. Thus, in the context of EPCC, the major drawbacks of the execution monitor of the VEP become negligible or even advantageous.

5. The VEP: 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 perform arithmetics correctly. In other words, the arithmetic instructions should have legal arguments. This security requirement is to avoid erroneous use of partial operators with arguments outside of their defined domain (e.g., division by zero).
- *Memory safety* prevents reading and writing to illegal memory locations. One can only read from the code space. The legal code space locations to read from are the locations with their address in $0, ..., N_c - 1$, where N_c is the code size. Even the instruction loading must be performed as legal reads from the code space.

  In the case of the heap, reads and writes are provided but in a very restricted manner. The only way to write a piece of data in the heap is to build a pair which has the data as its contents. All the pairs are legally built by the VEP and the type bit of the references to pairs is automatically set to 1 by the VEP. Likewise, in order to read (access) any data in the heap, the VEP provides instructions to access either of the two fields of a pair. Since the construction of the pairs is governed by the VEP, the programmer has no means to modify the type bit to *forge* a new pair and he has no means to read from the heap other than to access an existing pair.

In the case of the stack, reads and writes are permitted. Any read or write to the stack is preceded by a memory check which ensures that the read and write are going to be performed on valid stack locations as their destination. What a valid destination is varies from instruction to instruction. Generally, the valid read 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 just 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/or heap) to perform the instruction (e.g., the VEP raises an error if an attempt is made to pop when the stack is empty or to push an item onto a full stack).

- *Control-flow safety* prevents jumps outside of the code space.
- *Resource bound check* enforces limitations on the size of the code space, the size of the stack, the size of the heap, and the number of instructions the VEP may execute.
- *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.

6. The VEP: 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.

6.1. Initial Security Enforcement

The VEP checks the following requests for resources, only once, just before executing the code. Note that each request is made using a declaration in the header of the untrusted code. Each time, the VEP verifies whether the requested amount of resources is no greater than the maximum value settled in an agreement between the producer and the consumer.

- *Code size*: the producer inserts the demanded code size (dcs) of the proof generator in the header of the untrusted code. If the VEP refuses or fails to allocate the requested block of memory, the VEP refuses the untrusted code. Otherwise, the VEP allocates a block of dcs bytes of memory as the code space and inserts the code into the code space.
- *Stack size*: the VEP also checks the demanded stack size (dss) declared in the header of the untrusted code. If the VEP refuses or fails to allocate the requested block of memory above agreed-upon limit, the VEP refuses the untrusted code. Otherwise, the VEP allocates a block of dss words of memory as the stack memory.
- *Heap size*: the amount of demanded heap size (dhs) of the untrusted code is also mentioned in the code header. If the VEP refuses or fails to allocate the requested block of memory, the VEP refuses the proof generator. Otherwise, the VEP allocates a block of dhs words of memory as the heap memory.
- *Timeout*: the untrusted code should finish its task within a definite time period (i.e. number of operations). The demanded number of operations (dno) which is inserted by the code producer into the code header is checked by the VEP to ensure that it is not more than the agreed-upon limit. In the case where the requested number of operations is more than the limit the VEP refuses the proof generator.

6.2. Global Security Enforcement

When the code is not refused during the initial security enforcement, it is ready to be executed by 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 about to 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 as the number executed operations) has not exceeded the dno. If the number of executed operations is less than the approved number, then the check is passed, otherwise the code is refused for having run for too long.
- *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).

6.3. Instruction-wise Security Enforcement

The third level of security enforcement by the VEP is the fine-grained level and is done per instruction. By enforcing this level of security checks the untrusted code is prevented

to perform any unsafe operation. The instruction-wise security enforcement can be introduced by the means of the instruction called POKE. The POKE instruction allows one to write values in the stack. POKE pops p, pops q, then writes q to the stack location specified by p[1]. 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.

Figure 5 shows schemata of the stack in the VEP. The top schemata is a snapshot of the stack before the execution of the POKE operation. The human readable format of the same snapshot is shown on its left-hand side in which 0p represents the pair with address zero.

The POKE instruction needs two arguments. Therefore, the VEP first checks if there exist at least two elements on the stack. This check passes as there are 6 elements on the stack. As mentioned above, the VEP uses the argument p in order to calculate the destination in which the q has to be written. Therefore, the VEP checks if the type of p (the topmost element) is *number*. Here, the type bit of the top element (-4) is zero which indicates the type *number*. Next, the VEP checks if the calculated destination is a legal one. The legal stack destinations for writing are the ones greater than or equal to zero and smaller than or equal to SP. Here, p is negative, hence the destination, which is equal to 1 (SP+(-4) given that SP is 5), is a legal one. Since all these checks are passed successfully, the POKE operation is safe. The second schemata in Figure 5 shows the stack snapshot after the execution of the POKE instruction where the content of the stack location 1 is rewritten to 43.

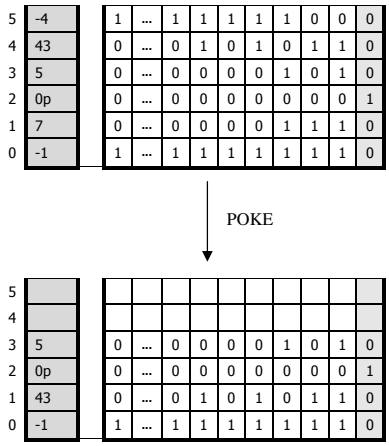

Figure 5. Instruction-wise security enforcement for POKE instruction

[1]The "push" and "pop" are not done incrementally and the stack pointer is updated at the end of the execution of the instruction. The POKE instruction first reads the top two elements of the stack: p is popped then q is popped. Then, it writes q to the stack location specified by p. Finally, it does two consecutive pops (i.e. it decreases the stack pointer by two cells).

Generally, after fetching each instruction and before the execution of the instruction, the VEP performs a combination of the following checks.

- *Number of operands*: the number of operands of an instruction can vary from zero to two implicit operands on the stack, depending on the instruction. For an instruction that requires with one or more operands on the stack, the existence of a sufficient number of operands must be checked before execution of the instruction. If insufficient operands lie on the stack, the execution is discontinued and the untrusted code gets refused.

- *Type of operand*: the VEP checks if the type of the operands conforms with the operation. As mentioned earlier, the values in the VEP can be numbers or pairs. The VEP can distinguish the type of an operand according to its type bit. Depending on the instruction and the operand, the latter may have to be a number, it may have to be a pair, or it may be free to be of either types. 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, the VEP ensures that the reads and writes have legal stack locations as their destination.

- *Sufficient memory*: the VEP verifies whether there is enough memory (stack and/or heap) to perform an instruction which works 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. It should be mentioned that the VEP does not enforce the concept of instruction boundaries. Therefore, the VEP can accept intertwined codes.

The complete set of instructions (32 kinds of instructions) along with their safety checks can be presented using a simple table. Due to lack of space, we do not present it here (see [27]). In this way, it would be an easy task to verify the safety of the VEP by pen and paper.

7. The VEP versus other VMs

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 another 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) [19] introduced in 1995 by Sun, and the .NET platform (CLR) [20] 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 the 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 (these large TCBs were the motivations for introducing the PCC approach in the first place). Appel *et al.* [3] measured the TCBs of various Java virtual machines at between 50,000 and 200,000 lines of code. The TCB size in these 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 ineffective PCC framework.

For EPCC, we need a virtual machine so simple that, it is feasible for a human to inspect and verify it by hand. None of the mentioned virtual machines and any other that we are aware of have been developed with this goal. JVM, .NET, and other well-known virtual machines are mostly focused on the performance, portability, etc. The implementation of the VEP is 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 it is simpler than popular virtual machines.

8. 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 a small number of lines it is feasible for a human to inspect and verify it.

Furthermore, in principle, the VEP does not need to increase the size of the TCB as it would be possible (without difficulty) to prove it safe in a PCC framework. Though we have not focused on proving the safety of the VEP through a PCC framework, the small size of the VEP makes it suitable for such work.

9. Sample Use of EPCC

In order to test the practicality of our approach, we have constructed a prototype implementation of an EPCC framework. Figure 6 shows the implemented framework in which the PCC framework is extended by employing EPCC. To complete the end-to-end chain we needed to implement 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 implemented an assembler for the VEP that allows a programmer to use instruction mnemonics instead of opcodes. Writing the proof generator program in a high-level language is even more convenient. 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 the C language. A detailed diagram of our sample implementation is presented in Figure 7. Next, we briefly talk about these three components, then we present an overall view and the detailed diagram of the implemented framework.

The VEP assembler translates source files written in the assembly language into the VEP language. Our assembler permits *assembly-time* arithmetic operations to take place in order to compute constants to include in the assembled program. Thus, the expressions are evaluated during the as-

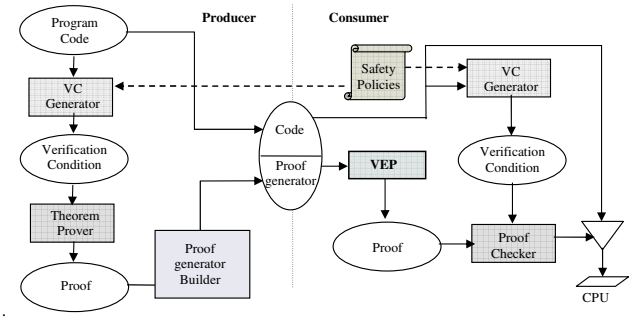

Figure 6. The EPCC version of the PCC framework

sembly 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 C compiler that we implemented is simpler than a complete C compiler because we used it to compile a specific application written in a subset of C language. Thus, we used a free yet incomplete C89 complier and implemented the two big and important phases of type checking and the code generation

The safety proofs in PCC are represented in the Edinburgh Logical Framework (LF) [15]. The typical LF representation of the proofs is 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 decompressed using the corresponding decompressor. 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 7 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 [11] is a popular data compression program that is based on the `DEFLATE` [10] algorithm, which is a combination of Huffman coding [16] and LZ77 [34].

In Figure 7, "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, we modified the extracted source code 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 `gotos`[1]. Since the computed `goto` is not supported by the ANSI C89 grammar, we added it to the grammar.

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. Calls to these custom functions are handled specially by our compiler and are translated into the VEP assembly language.

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. The compressed stream is then read by the decompressor 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. It takes us only a few minutes to build a significantly small proof generator from a new proof.

In the detailed diagram presented in Figure 7, 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.

In this sample implementation, we provide the producer with the possibility of sending a proof generator. This gives a chance to the producer to build a compact and special-

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

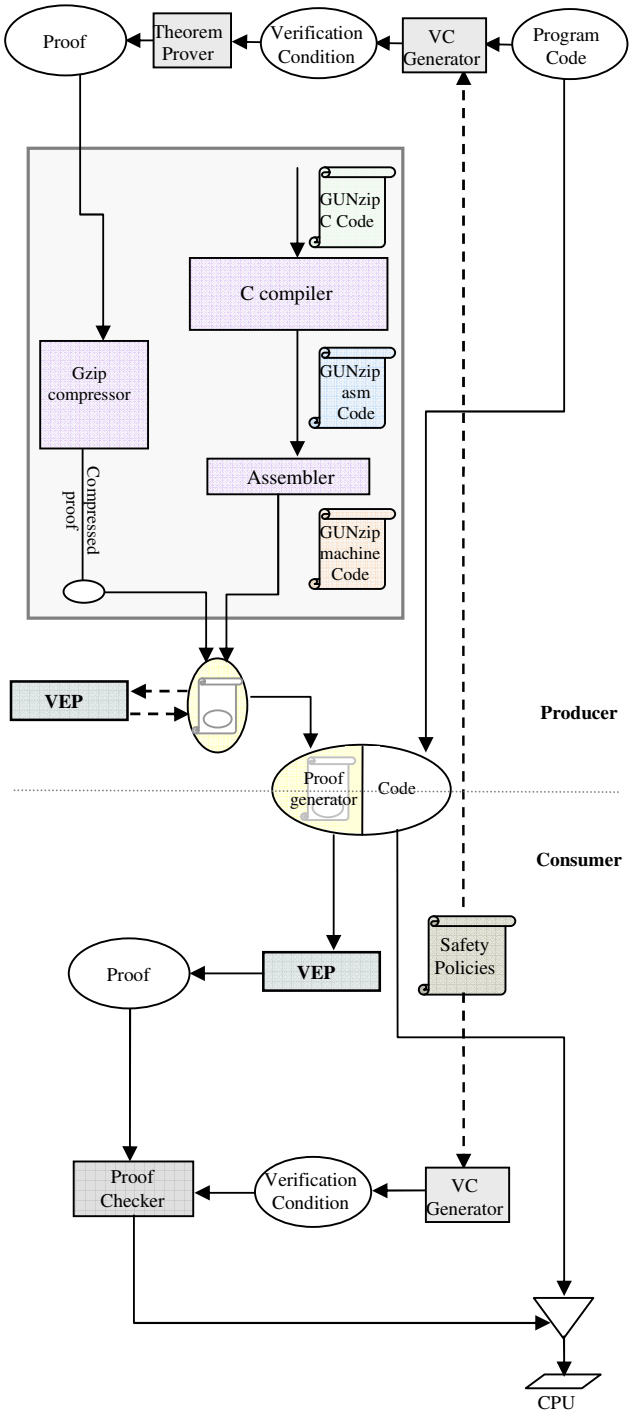

Figure 7. Detailed diagram of our sample implementation of EPCC

ized 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.

10. Future Work

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

Clearly, a first practical step will be to apply the framework to other possible PCC frameworks like FPCC and OPCC. 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 proof witness 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 would 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.

Another area of future work is to prove the VEP safe in a PCC framework. In this way, the VEP would not increase the size of the TCB at all.

One other direction to extend our approach is writing a proof generator which can translate a proof from a higher-order logic 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.

11. Conclusion

We described in this paper an extended version of the Proof-Carrying Code framework that offers a solution to the PCC's scalability issue.

We have worked on a hybrid approach, in which, instead of transmitting a proof, a proof generator for the code in question is sent. The new extended framework enables the execution of the proof generator on the consumer side in a secure manner on a safe and small virtual machine (the VEP). We showed empirically that the EPCC and its conjoint virtual machine the VEP have the potential to be used in an industrial-strength framework by implementing the necessary programs to complete the end-to-end chain. The fact that proofs contain many repeated patterns of proof rules and redundant arguments, makes them suitable for data compression. Our results show that proof generator

composed of an off-the-shelf decompressor and the compressed proof is significantly smaller than the entire proof (i.e. 20 times smaller). This allows us to scale PCC to checking the type safety of programs up to several hundreds of thousands of lines of code.

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. 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 asks you to believe very little and gives the highest priority to the security.

References

[1] A. W. Appel. Foundational Proof-Carrying Code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 247, Washington, DC, USA, 2001. IEEE Computer Society.

[2] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *CCS '99: Proceedings of the 6th ACM conference on Computer and communications security*, pages 52–62, New York, NY, USA, 1999. ACM.

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

[4] R. V. Binder. Six Sigma: Hardware Si, Software No!, 1997.

[5] CA Inc. CA Inc., 2008 Internet Security Outlook Report. White Paper, January 2008.

[6] J. Cheney. First-order term compression: Techniques and applications, 1998.

[7] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *PLDI '00*, pages 95–107, New York, NY, USA, 2000. ACM Press.

[8] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The Case for Virtual Register Machines. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 41–49, 2003.

[9] Department Of Defense Standard. Department Of Defense Trusted Computer System Evaluation Criteria, 1985.

[10] L. Deutsch. Deflate compressed data format specification.

[11] P. Deutsch. Gzip file format specification version, 1996.

[12] N. E. Fenton and M. Neil. A Critique of Software Defect Prediction Models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, 1999.

[13] P. L. George C. Necula. Proof-Carrying Code. Technical Report CS-96-165, CMU, Computer Science Department, September 1996.

[14] R. Grimm and B. Bershad. Security for Extensible Systems. In *HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 62, Washington, DC, USA, 1997. IEEE Computer Society.

[15] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[16] D. Huffman. A method for the construction of minimum redundancy codes. volume IRE 40, pages 1098–1101, September 1952.

[17] J. Joyce. Is error-free software achievable? *j-DATAMATION*, 35(4):53–56, February 1989.

[18] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. Technical Report YALEU/DCS/TR-1223, Dept. of Computer Science, Yale University, New Haven, CT, Jan 2002.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, second edition, 2000.

[20] E. Meijer and J. Gough. Technical Overview of the Common Language Runtime, 2000.

[21] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[22] G. C. Necula. *Compiling with proofs*. PhD thesis, Pittsburgh, PA, USA, 1998. Chair-Peter Lee.

[23] G. C. Necula. A Scalable Architecture for Proof-Carrying Code. In *FLOPS*, pages 21–39, 2001.

[24] G. C. Necula. *Advanced Topics in Types and Programming Languages*, chapter 5, pages 177–220. MIT Press, 2005.

[25] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI '96*, pages 229–243, New York, NY, USA, 1996. ACM.

[26] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL '01*, pages 142–154, New York, NY, USA, 2001. ACM.

[27] H. Pirzadeh. Encoding the program correctness proofs as programs in Proof-Carrying Code. Master's thesis, April 2008.

[28] H. Pirzadeh and D. Dubé. VEP: a Virtual Machine for Extended Proof-Carrying Code. 2008. To appear in *VMSec'08*.

[29] S. P. Rahul and G. C. Necula. Proof Optimization Using Lemma Extraction. Technical report, Berkeley, CA, USA, 2001.

[30] N. T. A. Reeves Glenn E. The Mars Rover Spirit FLASH anomaly. *IEEE Aerospace Conference*, 2005(5):4186–4199, March 2005.

[31] J. Saltzer and M. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975.

[32] F. B. Schneider, G. Morrisett, and R. Harper. A Language-Based Approach to Security. *Lecture Notes in Computer Science*, 2000:86–101, 2001.

[33] D. Wu, A. W. Appel, and A. Stump. Foundational proof checkers with small witnesses. In *PPDP '03*, pages 264–274, New York, NY, USA, 2003. ACM.

[34] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.