Using Pit to Improve Security in Low-Level Programs

Leif Pedersen <bilbo@hobbiton.org>
Hassan Reza <reza@cs.und.edu>
University of North Dakota
School of Aerospace Sciences
Department of Computer Science
Streibel Hall Room 201
3950 Campus Road Stop 9015
Grand Forks, ND 58202-9015 USA

Abstract. Pit[15][16] is a new language for low-level programming, designed to be a self-hosting alternative to C. The novelty is it supports automated memory management without excluding manual memory management, and without hindering key features associated with low-level programming, such as raw pointers, inline assembly code, and precise control over execution.

This paper presents Pit as a language, then examines how Pit's approach to memory allocation can be used to significantly increase the security of low-level programs. Automatic memory allocation is a useful tool of abstraction in many situations. Since Pit provides this tool without hindering low-level programming, it allows automated memory management to be used in programs where it previously could not be used, such as kernels. Specifically, this tool of abstraction can assist the programmer significantly in writing low-level code with fewer security problems caused by buffer overrun or integer overflow bugs by reducing the number of opportunities for such bugs in areas of code that do not need the precision of manual memory allocation. Existing solutions, such as Cyclone[23] add various ways of checking bounds, but have two major disadvantages: they require extra work from the programmer, and they detect but do not fix memory allocation bugs. Pit's approach simplifies what the programmer writes, making code more understandable.

1 Introduction

General-purpose programming languages, referring to languages designed for writing wide ranges of applications such as C, Perl, and their relatives, may be characterized by their style of memory management and level of abstraction. As a general trend, less abstract languages require the programmer to explicitly code memory management logic and statements express CPU instructions more directly with greater detail, while more abstract languages require less attention to memory management and represent the intent of the program more directly with less detail. There are, of course, many points of view on classifying languages, but for this discussion, the interesting trade-offs between general-purpose languages are manual or automatic memory management, representing CPU instructions directly or the intent of the algorithms, and representing more detail or less detail. For example, C represents the CPU's instructions almost directly and provides no automated memory management and requires more detail in the code, however Perl manages memory entirely automatically and usually represents complex algorithms in less code. The cost to using Perl is, of course, efficiency; in many cases, Perl code is so slow it cannot meet the user's requirements. This pattern of trade-offs frequently applies to comparisons between a low-level and a high-level language. This brief discussion may seem to imply a trade-off between automated memory management and expression of fine-grained detail, however even if this connection holds (and it may not) it does not exclude the possibility of creating a language that can express both sides of the trade-offs in adjacent statements. This duality is precisely what Pit implemented.

Unfortunately, most software does not divide as neatly as most languages into "low-level" and "high-level" where, for example, low-level applications cannot afford the cost incurred by high-level languages and high-level applications cannot afford the cost of extra effort in development time incurred by low-level languages. Most nontrivial software has some code that does not need to execute efficiently and therefore would benefit from a high-level language and some code that must execute quickly or for other reasons cannot afford the costs of a high-level language. Graphical programs are an easy example of this; code that deals with individual pixels and polygons for rendering images and animations benefits

significantly from optimizing at the level of individual CPU instructions, while code that directs larger user interface widgets such as window placement and buttons benefits far more from greater abstraction.

It's possible to bridge this gap by using two languages to write a program, but this approach has disadvantages. One problem is it's difficult for a module written in one language to access data stored in memory by a module written in a different language because most languages store data in memory quite differently from each other. Perhaps more importantly, using two languages to implement the software creates an artificial boundary in the logic based on the capabilities of the two languages rather than based on a natural separation of concern that's convenient for the program's design. A better solution is to use a language that allows both approaches and allows fine-grained control over when each approach is applied. Pit uses variable declarations to determine which approach to use, which allows convenient yet precise control.

Pit is a compiled language derived primarily from C. While it is designed to be familiar to C programmers, the languages are different enough in coding style that it will probably always be a manual task to translate between C code and Pit code in a way that produces reasonably understandable code. The novelty is that it simultaneously supports the variable types found in C, called "primitive variables" and another class of variables called "auto variables". Auto variables are declared syntactically using the keyword "auto" in place of the type specification. Auto variables are similar in many respects to variables implemented by Perl; in this regard, Pit is secondarily derived from Perl. They are dynamically typed and manage memory automatically. The programmer can choose to write the entire program using only auto variables, only primitive variables, or a combination of both. This allows the programmer to choose to use automated or manual memory management with every variable declaration. Thus, the programmer can write some code that has all the characteristics of C code by avoiding auto variables, and in the same function write more code that has all the benefits of automated memory management and dynamically typed variables by using auto variables.

Pit (named simply for Devpit.org) began as a classroom project to implement a C library for implementing dynamically-typed variables in C, which eventually developed into Pit's concept of auto variables. Its API used a single struct pointer type for variables which could store any type of value. The struct was opaque to the programmer, but indicated whether it contained an integer, string, hash, etc. We analogized this variable type to a Perl scalar variable. The API provided functions for the programmer to call for each operation, such as assign, add, reference, dereference, etc. Much like with Perl scalars, each variable carried a reference counter, which, for example, incremented when storing the variable in a larger structure. The programmer had the responsibility of calling a function which would decrement this reference counter or free the variable (recursively for multilevel data structures) every place execution could exit the scope containing the variable. This wasn't bad as far as complex data structure manipulation in C goes, but it was far too clumsy for managing simple strings; nobody writing "real" software would want to use this for simple things. We finally redesigned this concept as a new language. This has several advantages. Most importantly, it simplifies the concept of auto variables enough that they are easier to use than primitive variables, which are still as easy to use as C's variables. now the compiler automatically inserts the calls to decrement the reference count or free these variables; this tips the balance, making it easier to use than manual allocation. Further, we could now add several important features missing from other low-level languages such as exceptions and namespaces. Also, the programmer can write operations on the new variable type with symbolic operators in the same way as traditional variables, rather than as function calls. In adding these features, we kept the language within constraints that make suitable for the lowest-level programs traditionally written in C, especially kernels. To be sure, other languages have these features, but none of them fit within constraints these constraints. During this work, we studied ways that these improvements could be used for better security, portability, general maintainability, etc in low-level programs.

The compiler implements auto variables by inserting function calls to a supporting library. Each operation translates into a function call, and it inserts additional calls for incrementing and decrementing reference counts for garbage collection. It also transparently inserts calls to this library for converting from a primitive variable type to an auto variable or vice versa, making it easy to assign a value from a primitive variable to an auto variable and vice versa. Pit is self-hosting, meaning that it can express its own compiler tools and supporting libraries with no tools written in other languages (although this work is in progress). To accomplish this, the supporting language library uses primitive variables to construct the data structures that store information for auto variables.

Pit is designed to be suitable for (but not limited to) low-level programming, such as for operating system kernels, device drivers, system libraries, etc. This is perhaps the class of software that is most starved for improvements in languages because most implementations of recent languages require a virtual machine, prevent manual memory management, and don't have a way to access assembler code. Pit can interact with C code reasonably easily, which makes transitioning or interacting with an existing project easier. All C variables map directly to primitive variables in Pit, and the calling convention for functions is close enough that a simple wrapper function can work around the differences.

2 Examples of Pit syntax and translation to assembly

If auto variables could only store simple values, such as integers, floats, and references, they would obviously not be particularly useful. Their power becomes apparent when they are compounded to create a complex data structure. Creating a complex data structure using auto variables is trivial. To do so, simply assign an array or hash to an element of an array or hash, as follows.

```
1.  ...
2.  private auto $root;   // Declare $root as a local variable
3.  $root = array{};      // Assign an empty array to it
4.  $root(0) = hash{};    // Assign an empty hash to the zeroth element
5.  $root(0)("key0") = "array element 0, hash element key0";
6.  $root(0)("key1") = "array element 0, hash element key1";
7.  $root(1) = hash{};
8.  $root(1)("key0") = "array element 1, hash element key0";
9.  $root(1)("key1") = "array element 1, hash element key1";
10. ...
```

Notice how concise this code is because there is no need to explicitly tell the compiler how much memory to allocate for the array or hash tables, when to free the memory, or what the type of each value is. On line 2, we create an array and assign it to $root, which dynamically assumes the array type. The memory for the array is initially allocated with a size of zero, and later it is extended as necessary. On lines 3 and 6, we create hashes and assign them to elements of the array $root. Since the array was previously too small to hold the elements, it is automatically extended to a length of one, and then to a length of two. Similarly, the elements in the hashes need not exist before a value is assigned; they are created automatically as necessary. For comparison, here is similar logic using only primitive variables.

```
1.  ...
2.  private array(struct("key0" array(char, 64), "key1" array(char, 64)), 2) $root;
3.  $root(0)("key0") = "array element 0, hash element key0";
4.  $root(0)("key1") = "array element 0, hash element key1";
5.  $root(1)("key0") = "array element 1, hash element key0";
6.  $root(1)("key1") = "array element 1, hash element key1";
7.  ...
```

Notice that unless we invest a lot of time and code in manually allocating the memory with a dynamic size, we lose the ability to extend the array dynamically. Also, the structs are not extensible at all, and there are several opportunities for a buffer overrun vulnerability if the input strings come from a distrusted source. These are the same problems that all C code must deal with. However, the striking similarity between the two examples shows two things: first, that choosing between the two approaches is easy for the programmer; second, if the programmer uses the above implementation, an optimizer will be able to eliminate much of the inefficiency by converting patterns found in the first example into patterns shown in the second example when possible.

On the topic of efficiency, the array extension is not as time-consuming as it looks. Behind the scenes, the amount of memory reserved for the array starts at enough for four elements and is doubled every time it runs out of space. This reduces the amount of time spent copying the array to a new location in memory. While this is logically irrelevant to the program's execution, it reduces the time complexity of the extension from $O(n^2)$ to $O(n * \log(n))$, where n is the number of times one element is added to the array. Additionally, in many cases an optimizer could notice or guess how many elements are needed and pre-allocate the necessary amount of memory. In this case, it would not be hard for an optimizer to transform the entire example into primitive variables by using an array of structs. While beyond the scope of our

current work, an optimizer will eventually be an important tool in mitigating the cost of using auto variables instead of primitive variables.

Above, we mentioned that auto variables are implemented by translating every operation into a function call to a supporting language library. More precisely, for each auto variable, the compiler allocates a pointer where the variable's value would otherwise be stored; then each operation on it is translated into a function call so that a run-time library can manage its memory, type, value, ref-count, etc appropriately. For example, if an auto variable is declared in a function, the compiler allocates a pointer in the stack frame and a function call is inserted to allocate a glob of memory on the heap with the initial value of undef (undef is described later). Further operations on the variable result in the compiler inserting more function calls. At the end of the function, another function call is inserted to decrement each auto variable's ref-count or free it if appropriate. This way, it is valid to store a reference to the auto variable outside the function, since the glob need not necessarily be freed immediately when the function returns. Here is the exact translation for part of the example above into intermediate code for a 64-bit machine:

```
1.  // "$root" above is translated into "%base - 64b" here, which was already
2.  // allocated in the stack frame.  After each group of statements, the
3.  // stack is back to its initial state.
4.
5.  // 1.  private auto $root;
6.  stack_alloc 64b;                     // Subtract 8 bytes (64 bits) from
7.                                       // the stack pointer
8.  call $lang.auto.new.undef;           // Constructs a new auto with the
9.                                       // value undef and ref-count of 1
10. store <ptr> [ %base - 64b ], <ptr>;  // Pops the pointer to the new auto
11.                                      // and stores it at %base - 64b
12.
13. // 2.  $root = array{};
14. stack_alloc 64b;
15. call $lang.auto.new.array;           // Constructs a new auto with the
16.                                      // value of an empty array
17. store <ptr>, <ptr> [ %base - 64b ];  // Pushes the location of $root
18. call $lang.auto.refcount_inc;        // Inc ref-count because a binary
19.                                      // operator call decrements the
20.                                      // ref-count of both inputs
21. call $lang.auto.binop.assign;        // Perform the assignment, store the
22.                                      // result (a temp auto holding the
23.                                      // return value of the assignment)
24.                                      // in the second parameter's position
25. stack_free 64b;                      // Discard pointer to $root
26. call $lang.auto.refcount_dec;        // Free the temp auto
27. stack_free 64b;                      // Discard pointer to the temp auto
```

Although the translation looks somewhat obfuscated, it is no more so than the assembly output from a conventional C compiler, and this is easy output for a compiler to generate.

Notice that since a glob may be pointed to in multiple places if references to it are created, and therefore it cannot be reallocated with a larger size (because this may require relocating the glob). This means that the glob's value element must store a pointer to the value rather than the actual value, since the value must be reallocated if it changes in size. This extra layer of indirection allows, for example, string buffers that are stored in auto variables to be extended automatically as necessary rather than allowing a buffer overrun. It also allows for certain improvements to efficiency that we describe later.

3 Improving security

Automated memory management typically found in more abstract languages has benefits beyond saving significant amounts of work. It can entirely prevent some classes of security holes, such as buffer overrun[2] or integer overflow[5][1] bugs.

3.1 Related work

General concern over computer security is probably as old as wide-spread use of computers for routine tasks. In 1972 James Anderson published a report for the Electronic Systems Division of the US Air Force[3]. The primary concerns of his report were security between users of time-sharing systems and

security between computers on a network. For its age, his report is surprisingly relevant to modern computing. Security between users is at least as relevant today as it was in 1972 (even if multiuser machinery has perhaps declined), because most operating systems use it to minimize the threat that applications pose to each other. For example, a server running Apache and MySQL typically executes them with limited privileges as separate users so that if Apache is compromised, MySQL is still protected. The importance of security between computers has grown explosively as the Internet has grown, because nearly all computers are now connected to the Internet. However, it wasn't until the 1990s that the industry widely acknowledged that the numerous subtle security flaws in software affect everyone, not just organizations with especially sensitive secrets such as the US Air Force. Today, security is such a high priority that many users will accept severe performance penalties (especially in situations where faster hardware can compensate) or use software that is significantly inferior in other ways. On the other hand, there is significant successful work on reducing the cost of security, where cost refers to every trade-off – performance, maintainability, capabilities, hardware, etc.

Solutions for minimizing security holes in low-level software are still relatively sparse, and many common languages and programs predate wide-spread concerns about security. It wasn't until 1988 that the first documented exploitation of a buffer overrun appeared in the form of the Morris Worm. Interestingly, the report from Cornell on this incident[10] mentions mutual trust between computer users up to this point, and a desire not to "build walls as high as the sky" to protect against intruders. In today's environment of persistent identity thieves, spammers, bot-nets, pirates, and costly hoaxes it seems strange to look back on a time when we were so unconcerned with security. Looking back, it seems obvious that the Morris Worm was a disaster waiting to happen, however at a time before online shopping and personal computers for everyone raised the stakes, it probably wasn't unreasonable to assume attackers could be tracked down and prosecuted the same way they are with physical security breaches.

In 1996, the community's equilibrium was again punctuated by Aleph One. This time, there was no turning back. Computers had permeated the public to the point where too many people were interested in breaking security to ignore the issue any longer, and now Smashing The Stack For Fun And Profit[2] detailed exactly how to take advantage of unchecked buffers with an explanation most novice hackers could understand. This is roughly when intense work started in securing software against dedicated attackers and setting up ways of quickly responding as new security holes are discovered. The pattern of the buffer overrun programming bug has been astonishingly pervasive and persistent. In spite of 12 years of auditing since Aleph One threw open the gates, old buffer overrun bugs are still frequently discovered, and new ones are frequently introduced. Simply learning to write bug-free code has proven to be no more effective at eliminating buffer overrun bugs than at eliminating bugs in general.

Starting roughly 2002[5][1], attackers started exploiting integer overflow bugs and soon other non-control-data attacks[6]. Exploiting these bugs is a more subtle exercise and varies more between programs, but as it becomes harder for attackers to find buffer overrun bugs, they will undoubtedly focus more on other bugs.

Many tools have been developed that focus on detecting buffer overrun bugs at run-time. These can help developers fix old code and mitigate security risks, but none of the run-time tools can detect a bug without a test case that triggers it, and several types of errors are not reported by any of them, including integer overflow bugs and buffer overrun bugs within a data structure. Dynamic Buffer Overflow Detection[19] surveys some tools in this category including Valgrind[25], CCured[24], CRED[27], ProPolice[29], StackGuard[7], TinyCC[30], Chaperon, and Insure++. Safe Systems Programming Languages[12] surveys SafeC[28], CCured[24], Vault[26][31], and Cyclone[23].

Cyclone[23] represents good work, but does not provide a complete solution. It extends C by tagging pointers syntactically in different ways to provide bounds-checking. However, to take full advantage of its features requires modifications to the code that are so extensive that it is no harder to simply rewrite the code in a new language. Also, the programmer still has to specify how much memory is required, and Cyclone aborts the program when an allocation bug is detected rather abstracting out memory allocation entirely. This protects the system from intrusion, but does not actually fix any bugs. Often, aborting execution is not an acceptable solution. Further, Cyclone does not address integer overflow bugs. Writing code in Cyclone is more confusing than writing code in C because it provides several tags for pointers and the programmer needs to know which tag to use in each situation. Writing code in Pit with auto variables is simpler and easier, and actually fixes these bugs. Cyclone provides optional garbage collection, but it uses mark-and-sweep techniques, which exclude many important programs including kernel code and time-critical code. (Pit uses reference-counting garbage collection.) In Cyclone, the

programmer must still explicitly calculate the amount of memory required for allocations; automating this is a key ingredient of automated memory allocation.

SafeC[28] similarly protects against buffer overruns with run-time bounds checking, but sacrifices performance to avoid code modifications. Vault[26][31] is an entirely new language that does not support pointer arithmetic, which makes it unsuitable for kernel code. It also suffers from the disadvantage of requiring the programmer to explicitly track memory allocations while restricting pointers so tightly that it could have automated memory management entirely.

SmashGuard[14] presents a solution that is interesting because it is implemented in hardware so it requires no modification to application code and incurs no overhead. Its disadvantages are it protects only against attacks that target function call return addresses, it requires minor kernel modifications, and of course, requires new hardware.

Transparent Runtime Randomization[17] and Address Obfuscation[4] change the memory layout of applications with each execution. This is effective against most attacks because the attacker generally cannot predict the behavior of the program closely enough to compromise it. However, this does not protect against non-control-data attacks or potential denial-of-service attacks because of program crashes. PointGuard[8] is a tool with a different approach but similar results. Rather than rearranging the application's memory layout, it encrypts pointers when they are in memory so that a corrupted pointer will be decrypted to an unpredictable address before being accessed.

The methods that detect an attack in progress and abort the program do not fix the underlying problem that there is a bug in the software. Detection is a big improvement, and in most programs, this is a "safe" thing to do; although the program behaves in an undesirable way, disaster is averted. In many cases, however, aborting a program is dangerous. For example, an abort in the kernel will crash the computer, resulting in down-time and often lost data. A trapped bug in a database server can cause equally catastrophic results. An attacker may exploit any such bug to deliberately keep a service off-line. While sensitive data is not revealed, the solution is still not good enough.

Vault and Cyclone extend C but make code more complicated, and more complication typically means more errors. It is more desirable to simplify code through abstraction instead, and automated memory management is a powerful tool for this. Automated memory management also addresses non-control-data attacks and integer overflows, which most of the above methods do not consider. However, low-level developers have historically avoided languages that provide automated memory management because these languages also prevent access to raw pointers, in-line assembly, and necessary optimization techniques. (Automated memory allocation includes allocating and freeing memory without explicit statements. It also includes extending buffers to avoid overruns and widening integers to avoid overflows.)

This history suggests the following conclusions. C was designed before security was a major concern, and unfortunately in several ways it is impossible to extend C in ways that would solve the problem once and for all without significant changes to all the code in need of protection. The best way to solve memory allocation bugs is obviously to automate memory allocation, but before low-level developers can use automated memory allocation, there needs to be a language that supports both memory allocation and access to raw pointers. Such a language not only fixes the allocation bugs but also simplifies much code.

The primary disadvantage to creating a new language is that it requires rewriting code, but no solution has been presented yet that can fully protect a program without either unacceptable complexity, unacceptable cost to efficiency, or loss of access to raw pointers. Full protection includes protecting buffers that are part of a struct, and by design, C allows an array contained in a struct to be overrun, and many programs utilize that feature in reasonable ways.

3.2 Analysis

Even today, after so much study and work, cutting edge operating systems still have a dismal outlook on security. Even if all the bugs were fixed in existing code, new ones are constantly introduced with new code. For example, nobody reasonably expects the existing operating systems to ever be free of buffer overrun bugs until there are new innovations. Until then, the industry must settle for staying one step ahead of attackers. Operating system code is perhaps the most widely used code, since every computer needs it. This probably makes it the most targeted code for malicious attackers and probably the most widely audited code as well, but to date, nobody has produced a modern operating system that is free of security problems. OpenBSD is probably the operating system with the most proactive approach to security,

and after many years it is still unusual for them to go even a few months without finding another buffer overrun bug[13]. Many good ideas have been developed to improve security for low-level code, and yet all have fallen short of complete success, where a successful approach must achieve a high level of confidence that the code has no buffer overrun bugs, no integer overflow bugs, and do so without a loss of efficiency. In general, when this paper refers to security, the scope is limited to memory allocation, buffer overrun, and integer overflow bugs because they are the bugs that this programming language can help prevent, and this class of bugs covers the overwhelming majority of security holes.[18][9][8].

C is the language of choice for operating system code for good reasons. Low-level programmers absolutely need easy access to in-line assembly code for hardware IO and setting special-purpose CPU registers. Also, they need to be able to write efficient code, as speed is a major point of competition between operating systems; this means that if there are any inefficient high-level features, they need to be optional. (C has no such features) A language that requires an interpreter or virtual machine is clearly not an option. Perhaps most importantly, programmers need to have a clear idea what the output of the compiler will be, because they are working in an environment where many CPU facilities are often unavailable. When the translation is too abstract or layered too deeply, a low-level programmer will often become frustrated because the compiler will generate unexpected output that may, for example, interfere with the kernel's memory management, use floating-point facilities when they are not available, or attempt to access variables that are temporarily unavailable. Complex tools lead to more of these problems than simple tools of equal power. To address this, source code that does not use auto variables will never be translated into machine code that automatically manages memory. Because the extra features have no impact on the output unless they are used, Pit works at least as well as C for low-level code. Pit fits all of these requirements, and provides a way to choose when to use automatic memory management as well. Pit also interacts more easily with C than most other languages. This makes it a good step forward from C with little cost.

Pit is by no means a silver bullet. The advantage is that the programmer gets to choose whether the benefit is worth the penalty, rather than being forced to write an entire module in a low-maintenance high-overhead language such as Perl or in a high-maintenance low-overhead language such as C. To effectively use Pit, a programmer must choose wisely between primitive and auto variables. This requires a balance of efficiency and maintainability. As long as a programmer only uses auto variables, there is an absolute guarantee that his code will not have these security bugs. However, this guarantee comes at a cost to efficiency that is often significant. On the other hand, every time a programmer uses primitive variables, he runs the risk of introducing security bugs. It is well-established that program execution time is related to Zipf's Law. Specifically, a small amount of code accounts for most of the execution time of a program; a widely accepted estimate is that 10-20% of the code accounts for 80-90% of the execution time. (The sections of code that account for most of the execution time are the "time-critical sections" sections of code.) On the other hand, Zipf's Law does not apply to security. When considering execution time, a program can be somewhat less efficient, but a program cannot be somewhat less secure. It's either secure or not, because when an attacker discovers a security hole, typically the whole system is compromised. This leads to the conclusion that while all the code in a program must be secure, only about 10-20% of it needs to be efficient. It is easy to take advantage this in Pit. If only 10-20% of the code is written using primitive variables, then the risk of security holes is eliminated in 80-90% of the code. Additionally, this leaves programmers more time to carefully audit the remaining code for security bugs, both because they don't need to audit the other code for these bugs and because the other code was easier and faster to write initially. To benefit from this aspect of Pit, a programmer must to be willing to accept this trade-off and use it to his advantage. Most of the existing research for detecting security errors in C code or aborting a C program when a buffer overrun occurs is easy to adapt to Pit's primitive variables. Such an adaptation should typically assume auto variables have no such security errors and do the security checking only on primitive variables.

Perl has already proven this approach. When using Perl, programmers write most of the code in Perl, but call C libraries to do the small amount of work that consumes a large amount of CPU time. The Perl code cannot have buffer overrun bugs, (with the bigint extension) integer overflows, double-free errors, memory leaks, non-control-data attacks[6], etc. However, Perl's approach requires the awkward division of a program into high-level and low-level parts and is entirely unusable for writing low-level software, as discussed in the introduction.

3.3 Buffer overrun: C comparison

Simplifying code reduces human error. Human error is the only reason new buffer overrun bugs are introduced into code. We will use a short program written in C and a short program written in Pit to demonstrate how using Pit's auto variables simplifies code and eliminates the risk of buffer overruns by abstracting the logic for memory allocation out of the written code into the compiler and language library.

Figure 1 shows a C program that adds pairs of numbers read from standard input. Figure 2 shows the same program written in Pit. Notice that in Pit, auto variables can manage all the array sizes and memory allocation, thus eliminating any potential for mismanaged string buffers. As shown in Figure 3, they can also be managed manually just as easily as in C, with the same risk to security. This is the trade-off between improving efficiency and improving security and maintainability.

While Figure 1 has no buffer overrun vulnerabilities, there are several ways they could easily be introduced either when the code is modified or as a simple miscalculation when writing it in the first place. For example, the wrong size could be passed to fgets(), a pointer error could be introduced where the string is parsed with strsep(), or an error could be introduced in the format string passed to printf(). Also notice that the code sets an arbitrary limit on the length of an input line, and while it is obviously possible to work around this, doing so makes the program much more complex and adds more opportunities for security holes. For example, dynamically allocating just enough memory to hold the input can lead to accidentally freeing a dynamically allocated chunk of memory twice[6] or misusing realloc(). These are the very errors that are commonly found in C code, but never found in Perl code.

Figure 2 shows the same program, this time written in Pit using auto variables. Notice that neither the length of a string nor a pointer into the string are ever used, and the bounds on any indexing operations for auto variables are checked at runtime. All the sizes calculated automatically because memory is managed automatically. This means the opportunities for security holes that were present in Figure 1 are eliminated.

Figure 3 is an exercise in optimization, trading confidence in security for efficiency. It shows the same program written in Pit, this time using some primitive variables. Notice how primitive and auto variables easily work together, allowing the programmer to choose precisely what code to optimize for efficiency at the risk of security and maintainability, and what code to optimize for security and maintainability at the cost of efficiency. In particular, note that $in is implicitly converted to an auto variable when passed to $string.split with little effort on the programmer's part. This level of interaction is impossible when using two languages such as Perl and C together; instead, the programmer must write two separate modules and tie separate functions together. A programmer can use this, for example, to efficiently do time-intensive calculations of plotting a temperature graph from trustworthy input data in the same function that parses distrusted inputs strings from a web browser to choose color preferences or add decoration.

```
1.  #include <stdio.h>
2.  #include <string.h>
3.
4.  int main() {
5.      char in[1024], *inptr, *num;
6.      int first, second;
7.
8.      while(fgets(in, sizeof(in), stdin)) {
9.              inptr = in;
10.
11.             if(! (num = strsep(&inptr, " ")))
12.                     continue;
13.             first = atoi(num);
14.
15.             if(! (num = strsep(&inptr, " ")))
16.                     continue;
17.             second = atoi(num);
18.
19.             printf("Sum is %i\n", first + second);
20.     }
21.     return 0;
22. }
```

Figure 1. Example program in C.

```
1.  import io;
2.  import string;
3.
4.  public function(out int $exit_status) $.main = {
5.      private auto $in, $nums;
6.
7.      // $io.stdin is a global symbol.  => is
8.      // the method-call operator.  readline
9.      // is the name of the method that reads
10.     // a line of input.  The line is stored
11.     // in $in and returned.  Dots are
12.     // simply part of a symbol's name
13.     // indicating its namespace.
14.     while($io.stdin=>readline(ret $in)) {
15.             // $string.split() is a function that
16.             // splits $in on " " and stuffs the
17.             // result into $nums as an array of
18.             // strings.
19.             $string.split($in, " ", $nums);
20.
21.             // Convert the strings to integers.
22.             $string.2int($nums(0));
23.             $string.2int($nums(1));
24.
25.             // The object $io.stdout has a method
26.             // called writef that is semantically
27.             // analogous to C's printf().
28.             $io.stdout=>writef(
29.                     "Sum is $sum\n",
30.                     hash {
31.                             "sum", $nums(0) + $nums(1),
32.                     },
33.             );
34.     }
35.
36.     $exit_status = 0;
37. }
```

Figure 2. Rewrite of Figure 1 in Pit.

```
1.  import io;
2.  import string;
3.
4.  public function(out int $exit_status) $.main = {
5.      private string(char, 1024) $in;
6.      private int $in_len;
7.      private auto $nums;
8.
9.      loop: {
10.             // Pit strings are _not_ null-
11.             // terminated.  $in_len is an in-out
12.             // parameter.  Going in it's the
13.             // maximum length of $in and coming out
14.             // it's the actual length of $in.
15.             $in_len = sizeof($in);
16.
17.             // _p means primitive.  => is the
18.             // method-call operator.
19.             $io.stdin=>readline_p($in, $in_len);
20.
21.             // readline_p modified $in_len
22.             unless($in_len) break;
23.
24.             // Split works with autos, so this
25.             // type-cast is required to tell the
26.             // compiler that the string $in is
27.             // shorter than 1024 bytes.
28.             $string.split($in[string(char, $in_len)], " ", $nums);
29.
30.             // Convert the strings to integers.
31.             $string.2int($nums(0));
```

```
32.             $string.2int($nums(1));
33.
34.             $io.stdout=>writef(
35.                     "Sum is $sum\n",
36.                     hash {
37.                             "sum", $nums(0) + $nums(1),
38.                     },
39.             );
40.     }
41.
42.     $exit_status = 0;
43. }
```

Figure 3. Some speed optimizations to Figure 2.

3.4 Integer overflow

Integer overflow bugs are more insidious and are a much newer problem than buffer overrun bugs that have not gained notice until around 2002. The pattern of this bug is that the program will behave unexpectedly when an integer's value overflows during arithmetic. These overflows happen in two common situations, either while adding two positive or two negative numbers (or subtracting equivalently) or while multiplying. When adding a positive number to a negative number there is no problem, and when dividing there is no problem. (Code that calculates powers for memory allocation is much more rare, however that does not entirely exempt power calculations from concern. Multiplication is bad enough for our discussion.)

First, consider adding. Adding two positive numbers with an overflow produces a negative result. An easy example is overflowing while incrementing. While iterating over the characters in a buffer, if the counter variable overflows, the code will accidentally follow a negative offset into the buffer. This particular situation isn't terribly difficult to handle in a number of ways, but extending the example into multiplication shows a far more severe problem.

The resulting sign of a multiplication is not a reliable indicator of overflow. Multiplying two fairly small positive values may produce an overflow so large that the result wraps around more than once, producing a positive result. For example, multiplying 65536 by 65537 using 32-bit integers results in a positive value much smaller than expected. Multiplication is used in calculating the memory address of array elements. This is the root of the problem in the following example.

Just last year (in 2008), CUPS[20] announced a patch[21] for an integer-overflow bug leading to remote exploitation and local privilege escalation[22]. The problem was that the result of multiplying two integers and passing the result to calloc() could result in an allocation smaller than expected. In subsequent code, iterating over this allocation results in overrunning the allocation even though the counter variable stays within reasonable boundaries. If calloc() instead used an unbounded integer for its parameter there is no concern at all. This would allow calloc() to simply return an error indicating that it cannot allocate the requested amount of memory. The solution for this C code is an awkward check to be sure each input variable is not greater than 32767. Additionally, there is nothing to indicate to code maintainers that part of the reason for this bound is the size of the type lchar_t. Note that this class of problems and this solution can be extended to most functions that write to caller-allocated buffers, such as read(), fread(), recv(), fgets(), etc.

4 Future work

The first work to do next is to write a Pit compiler in Pit. This will both demonstrate that Pit can be self-hosting and improve on the current compiler, which has an overly simple design with no optimizations. It was written for a single purpose, which is to bootstrap the compiler written in Pit.

Then the compiler needs some optimization. Most of the optimization techniques used by C compilers should be useful in Pit. There are additional Pit-specific optimizations. For example, a primitive variable can often be substituted for an auto variable as described above. When work on optimization is complete, it is reasonable to expect Pit programs that use only primitive variables to match the speed of equivalent C programs. Pit programs will always be faster than equivalent Perl programs because when a program written in Perl executes, Perl first compiles it into byte-code, then executes it in a virtual machine. Pit programs will execute significantly faster, even if they only use auto variables, because the code is

already compiled at run-time and does not require a virtual machine. This places the performance of Pit programs, at worst, somewhere close to C programs and better than Perl programs.

Next, there needs to be some study of the difference in speed between primitive and auto variables, specifically to determine how high the performance penalty is for using auto variables. This will provide insight into when it's appropriate to use primitive or auto variables when weighing the trade-off of a program's performance. Some rough study on this could be done today on toy examples by emulating the logic for auto variables in C. The examples would be lengthy, inconvenient, and somewhat inconclusive, but with care they could serve the purpose of giving a rough estimate of the difference in speed between the types of variables.

A later useful branch of work may be to create a compiler that translates Pit code into a byte-code that is suitable for a virtual machine. Like Java, this would allow programs to be portable between CPU architectures. It is possible to do this in a way that allows many programs to be compiled natively or for the virtual machine without modification, except programs that use inline assembly code.

Another possible branch of work may be adapting existing research on checking C code for security bugs. While this is obviated for code that exclusively uses auto variables, this research may still be useful to minimize security risks in code that uses primitive variables.

5 Conclusion

Defining a new low-level language with specific features for memory management is a promising approach to improving the quality of low-level software. So far, research in the area of buffer overruns and integer overflows have focused on either detecting these bugs or adding features to C that specifically deal with these problems. Thus far, these approaches are either incomplete or make software much harder to write. Pondering the larger point of view of using automated memory management, which is a nonspecific tool that's useful in many ways for simplifying code in general, reveals a new point of view for preventing these problems.

Pit avoids infrastructure that usually deters programmers from choosing other languages for low-level code because of high overhead or unpredictable behavior, such as an interpreter, a virtual machine, or a mark-and-sweep garbage collector. There is no performance penalty to using Pit when the programmer chooses to use only primitive variables. When the programmer chooses to use auto variables, there is a cost (how much is yet to be seen). However because of the relation of Zipf's Law to execution time, the penalty of automated memory allocation will not be important in most of the code. When efficiency is necessary, Pit provides an exceptionally easy way to interface auto variables with optimized code. Perhaps the biggest advantage is Pit's ability to describe time-critical sections in efficient terms while protecting the rest of the code from buffer overruns, integer overflow bugs, non-control-data attacks, double-free bugs, and other memory allocation bugs. These considerations make Pit a superior language for low-level programming.

6 References

[1] Dave Ahmad. The rising threat of vulnerabilities due to integer errors. IEEE Security and Privacy, 01(4):77–82, 2003.

[2] Aleph One. Smashing the stack for fun and profit. Phrack http://www.phrack.org/issues.html?issue=49&id=14, 7(49):14, 1996.

[3] J. P. Anderson. Computer security technology planning study. Air Force Electronic Systems Division ESD-TR-73-51, Vols. I and II, 1972.

[4] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In 12th USENIX Security Symposium, August 2003.

[5] Blexim. Basic integer overflows. Phrack http://www.phrack.org/issues.html?issue=60&id=10, 11(60):10, 2002.

[6] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In 14th USENIX Security Symposium, pages 177–192, 2005.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and

prevention of buffer-overflow attacks. In 7th USENIX Security Conference, pages 63–78, January 1998.
[8] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuardTM: Protecting pointers from buffer overflow vulnerabilities. In 12th USENIX Security Symposium, August 2003.
[9] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In DARPA Information Survivability Conference & Exposition – Volume 2, pages 119–129, January 2000.
[10] T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro. The Cornell commission: on Morris and the worm. Communications of the ACM, 32(6):706–709, 1989.
[11] FreeBSD. FreeBSD ports collection. http://www.freebsd.org/ports/.
[12] Peng Li. Safe systems programming languages, October 2004.
[13] OpenBSD. OpenBSD security. http://openbsd.org/security.html.
[14] Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. Smashguard: A hardware solution to prevent security attacks on the function return address. IEEE Transactions on Computers, 55(10):1271–1285, 2006.
[15] Leif Pedersen. Pit programming language. http://pit.devpit.org/.
[16] Leif Pedersen and Hassan Reza. A formal specification of a programming language: Design of Pit. In ISOLA ’06: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006), pages 111-118, Washington, DC, USA, 2006. IEEE Computer Society.
[17] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. srds, 00:260, 2003.
[18] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture support for defending against buffer overflow attacks. In Workshop on Evaluating and Architecting System Dependability (EASY), October 2002.
[19] Michael Zhivich, Tim Leek, and Richard Lippmann. Dynamic buffer overflow detection. In Workshop on the Evaluation of Software Defect Detection Tools 2005, 2005.
[20] Common UNIX Printing System. Project home page. http://cups.org/
[21] Common UNIX Printing System. STR #2919: Multiple vendor CUPS texttops integer overflow vulnerability. http://www.cups.org/str.php?L2919 and http://cups.org/strfiles/2919/str2919.patch
[22] iDefense Labs. Public advisory: 10.09.08. http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=752
[23] Trevor Jim and J. Greg Morrisett and Dan Grossman and Michael W. Hicks and James Cheney and Yanling Wang. Cyclone: A Safe Dialect of C. In ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, pages 275-288, Berkeley, CA, USA, 2002, USENIX Association.
[24] George C. Necula and Scott McPeak and Westley Weimer. CCured: type-safe retrofitting of legacy code. In POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on principles of programming languages, pages 128-139, Portland, Oregon, 2002, ACM.
[25] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In Proceedings of RV'03, Boulder, Colorado, USA, July 2003.
[26] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. SIGPLAN Notices, 36(5): 59-69, 2001.
[27] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In In Proceedings of the 11th Annual Network and Distributed System Security Symposium, pages 159-169, 2004. http://citeseer.ist.psu.edu/ruwase04practical.html
[28] Todd M. Austin and Scott E. Breach and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. SIGPLAN Notices, 29(6): 290-301, 1994.
[29] H. Etoh and K. Yoda. ProPolice: Improved stack-smashing attack detect on. IPSJ SIGNotes Computer SECurity 014(025), Oct.2001. http://www.trl.ibm.com/projects/security/ssp
[30] TinyCC. http://tinycc.org/
[31] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. SIGPLAN Notices, 37(5): 13-24, 2002.