Recent Developments in Low-Level Software Security

Pieter Agten, Nick Nikiforakis, Raoul Strackx, Willem De Groef, and Frank Piessens

IBBT-Distrinet, Katholieke Universiteit Leuven, Belgium,
firstname.lastname@cs.kuleuven.be

Abstract. An important objective for low-level software security research is to develop techniques that make it harder to launch attacks that exploit implementation details of the system under attack. Baltopoulos and Gordon have summarized this as the *principle of source-based reasoning* for security: security properties of a software system should follow from review of the source code and its source-level semantics, and should not depend on details of the compiler or execution platform.
Whether the principle holds – or to what degree – for a particular system depends on the attacker model. If an attacker can only provide input to the program under attack, then the principle holds for any *safe* programming language. However, for more powerful attackers that can load new native machine code into the system, the principle of source-based reasoning typically breaks down completely.
In this paper we discuss state-of-the-art approaches for securing code written in C-like languages for both attacker models discussed above, and we highlight some very recent developments in low-level software security that hold the promise to restore source-based reasoning even against attackers that can provide arbitrary machine code to be run in the same process as the program under attack.

Keywords: software security, C language, full abstraction

1 Introduction

Programming languages are supposed to provide developers with a high-level abstraction of the platform on which programs will eventually be executed. The programmer should be able to reason about his code at source-code level, and let the compiler and run-time system worry about the low-level execution platform details.

Unfortunately, programming languages fail to do this from the point of view of security. Attacks against software systems often depend in an essential way on details of the platform on which the software is executed, and one can for instance not understand the security consequences of a bug in the program without understanding many details of the execution platform.

This is obviously the case for attacks that exploit memory errors in programs written in unsafe languages such as C and C++. Understanding low-level attacks such as stack smashing attacks, direct code injection attacks, jump-to-libc attacks or return-oriented programming requires one to understand many details of the compiler, operating system or processor architecture [14].

But it is also true for attacks against any software system – including software written in safe languages – where the attacker can interact with the program at the machine code level. For instance a malicious natively implemented function called from a Java program can attack the Java program in very powerful ways and such attacks wil again depend essentially on many details of the execution platform. In a similar way, a natively implemented browser extension can attack any web page visited, or a malicious kernel module can install a root kit.

An important objective for low-level software security research is to correct this situation, and restore what Baltopoulos and Gordon have called the *principle of source-based reasoning* for security [5]. It should be sound to reason about security properties of a software system on the level of source code. In this paper we discuss state-of-the-art approaches for both attacker models discussed above, and we highlight some very recent developments in low-level software security that hold the promise to restore source-based reasoning even against attackers that can inject arbitrary machine code.

The remainder of this paper is structured as follows. First, we illustrate low-level software attacks in both attacker models in Section 2. Next, we briefly discuss the state-of-the-art in securing C programs in the first attacker model where attackers can only provide input to the program. This is a well-understood problem with many mature solutions, and we provide an overview in Section 3. In Section 4, we turn to the more challenging attacker model, and we discuss two recent lines of research that make important steps forward against such attackers.

2 An illustration of low-level attacks

We distinguish two different but related attacker models. In the first model, that we call the *interactive attacker model*, an attacker can interact with the program under attack by providing input and reading output. An interactive attacker can for instance try to exploit a buffer overflow vulnerability if the program was written in an unsafe language, or could try to do SQL injection, or exploit a logic flaw against a program written in a safe language. The interactive attacker model is a reasonable model for the case where an attacker is trying to subvert a network service running on a hardened and well-protected server machine.

In the second model – the *in-process attacker model* – an attacker can load arbitrary machine code in the process executing the program under attack. This attacker can for instance scan memory for secrets, and overwrite control-flow data, or non-control data of the program under attack, even when the program was written in a safe language. The in-process attacker model is a reasonable model for the case where applications can be extended at run-time with (binary)

plugins, or for the case where an application is built from components coming from different stakeholders.

An interactive attacker against a program written in an unsafe language can escalate to an in-process attacker by doing a code-injection attack as we will discuss below.

2.1 The interactive attacker model

In this attacker model, the principle of source-based reasoning fails for unsafe languages such as C or C++. This is well-known and many papers give examples; we refer the reader to Erlingsson et al. [14] for an overview. Here is one simple example of a program in C for which the principle of source-based reasoning fails.

Example 1. In the presence of memory errors such as buffer overflows, an interactive attacker can modify variables in ways that can not be explained by the source code semantics, but that can only be explained by looking at details of the compiler and execution platform.

Consider the example vulnerable function `do_maintenance` of Code Listing 1. The purpose of the `do_maintenance` function is to read the username and password of the current user and if these credentials are valid then perform privileged operations. Source level reasoning can lead a programmer to believe that privileged operations can only be executed after a succesful call to `valid_credentials`.

However, the program has a buffer overflow vulnerability: the programmer has incorrectly used the size of the password buffer for reading in the username, thus allowing the attacker to overflow four characters past the username buffer. The source code semantics (in this case the C standard) says that further behaviour of the program is then *undefined*.

However, by relying on details of the compiler and execution platform, an attacker can perform a useful attack. Compilers will typically allocate the local variables of a function one after the other on the stack, and hence by overflowing the username, the attacker can modify the `authenticated` variable. Since any non-zero value for this variable will be interpreted as true, this will give the attacker access to the authenticated part of the program without the need of a valid username and password combination.

It is often the case that memory errors in a C program can allow an interactive attack to escalate to an in-process attack. The attacker achieves this by performing a so-called *code injection attack*.

Example 2. Code Listing 2 is vulnerable to a traditional code injection attack. The purpose of the program is to read a string from the user, perform a transformation on that string and then save it along with the original string, in an object-oriented programming style. Since there could be many transformations, the transformation function is called through a function pointer which is set

Code Listing 1 Code snippet vulnerable to a non-control data attack

```
int do_maintenance () {
    int authenticated = 0;
    char username[24];
    char password[28];

    fgets(username, sizeof(password), stdin);
    fgets(password, sizeof(password), stdin);

    if (valid_credentials(username,password) == 1)
        authenticated = 1;

    if (authenticated){
        //Do privileged operations
    }
}
```

by the programmer before the copying of the string. The code that reads the string from the execution environment is vulnerable to a buffer overflow since it doesn't perform any checks whether the `src` buffer is large enough to hold the contents of the command line argument. If the attacker provides a string that is longer than 128 bytes, the string will spill out to the `dst` buffer. If the provided string is longer than 256 bytes then the string will also overwrite the function pointer that is called in the next line. The attacker can simply enter his shellcode in the `src` buffer and overwrite the function pointer with the address of the buffer. Thus the program, instead of calling `capitalize`, will jump to the attacker-provided shellcode.

Again, the source code semantics would say that further execution of the program after overflowing of the buffer is undefined. But relying on many details of the compiler and execution platform (including the layout of variables in memory, and the fact that the program executes on a Von Neumann architecture where code and data are in the same memory) the attacker can actually have some of the data that he inputs to the program be interpreted as code. In other words, the interactive attack escalates to an in-process attack.

2.2 The in-process attacker model

In this model, an attacker is given the ability to load and execute code in the same process as the program under attack. For instance, when a user installs a plug-in for an extensible program, the plug-in will traditionally run in the same process. Also, as discussed earlier, an interactive attacker can inject code in a process running a vulnerable C program.

Against an in-process attacker, the principle of source-based reasoning fails completely. Consider for instance a browser that can be extended with new

Code Listing 2 Code snippet vulnerable to a heap-based buffer overflow

```
struct data_node {
    char src[128];
    char dst[128];
    int (*transform_func)(char *, char*);
};
int main (int argc, char *argv[]) {
    struct data_node *n;
    int i;
    n = malloc(sizeof(struct data_node));
    n->transform_func = capitalize;

    for(i=0; argv[1][i] != '\0'; i++)
        n->src[i] = argv[1][i];

    (*n->transform_func)(n->src, n->dst);
}
```

features and functionality by loading native plug-ins. If the browser stores secret information such as cryptographic keys or passwords in memory, then a malicious plug-in can find and read all these secrets by scanning memory, even in the case where the browser stored these secrets in variables or fields that – according to the source code semantics – should be private.

Whereas the problem of restoring source-based reasoning for the interactive attacker is well understood, the same problem for the in-process attacker is much more challenging. Making source-based reasoning sound in the presence of in-process attacks is very much an open problem. Recent advancements however, allow a system to maintain certain security guarantees even when an attacker can execute arbitrary code. These advancements will be explored in Section 4, and may open up the possibility to restore the principle of source-based reasoning even against in-process attackers.

3 Countermeasures against the interactive attacker

Low-level software vulnerabilities in the interactive attacker model are essentially bugs in the program that allow an attacker to drive the program into a state where – according to the source programming language semantics – further behaviour of the program is *undefined*. In practice, this means that further behaviour depends on compiler, runtime system or operating system details, exactly the kind of thing that the principle of source-based reasoning argues against.

For the interactive attacker model, this is a well-understood and widely studied problem. Broadly speaking there are two types of solutions.

3.1 Safe languages

From a programming language point of view, defenses against the interactive attacker are well understood. A programming language is *safe* if – informally speaking – it is completely defined by its programmer's manual [23]. This is of course just another way of phrasing the principle of source-based reasoning. Technically, safety is achieved by ruling out dangerous language features, and through a combination of compile-time and run-time checks. The objective is that any bug that could lead to implementation-dependent behaviour (such as accessing an array out of bounds, or dereferencing a dangling pointer) is either impossible to write in the language (e.g. dangling pointers do not exist in a language with automatic garbage collection), will be detected at compile time (e.g. casting an integer to a reference will be prohibited by the type checker), or will lead to well-defined error behaviour at run-time (e.g. throwing an exception on accessing an array out of bounds).

Many modern languages are safe, or at least provide very restricted access to unsafe features. Examples include Java, C#, Haskell, Scala and so forth. There is also a significant body of research on designing languages that are safe, but try to stay very close to C, so-called *safe dialects of C* [17, 22].

Despite this important progress in language design, most software engineers do not expect the C language to disappear any time soon, and hence the proposal of new languages is only a partial solution.

3.2 More defensive execution of unsafe languages

A wide variety of techniques has been developed to execute unsafe languages more defensively [14,34,35]. Roughly speaking, these techniques can be grouped into two categories.

Additional run-time checks: the idea here is to detect source-level undefined behaviour by means of run-time checks and to terminate the program. Proposed techniques range from simple heuristics, such as *canaries* to fairly complete *bounds-checking*.

The concept of canaries as a means of protection was first used by StackGuard [12]. StackGuard added a new random value on the stack between the return address and stored stacked pointer which the function checked before using the stored return address. If the canary was modified, that was a sign of a buffer overflow and thus the program was terminated before the possibly-modified return address was used. ProPolice [15] later re-implemented StackGuard and added a series of new features that increased the overall security of the stack, e.g. re-organizing the local variables and placing character buffers right next to the canary. ProPolice-like countermeasures are widely used in modern operating systems. Variations of the canary-principle have also been proposed to protect a program's heap [24, 37] and individual program variables [31].

In bounds-checking, countermeasures attempt to give to the C and C++ programming language what they, by design, lack: memory safety. These countermeasures insert additional bounds-checks before critical operations. Depending on the frequency and types of checks, these checked programs can be significantly slower than their unchecked versions. Even though the latest proposed bounds checkers [3,36] are many times faster than their older versions [18], they still impose a non-negligible performance overhead on running systems.

Finally, also techniques based on memory protection, such as setting data memory to be non-executable can be seen as additional run-time checks to terminate a program that has run into source-level undefined behaviour. Many modern operating systems will make for instance the stack and/or heap non-executable.

Randomizing or obfuscating execution platform details: The most popular instantiation of this principle is Address Space Layout Randomization (ASLR) which is currently implemented in all modern operating systems [7]. When a process is fully protected with ASLR, its stack, heap and libraries are always loaded in different memory offsets. The rationale is that even if an attacker can trigger a memory error, and hence has the power to overwrite program variables or hijack the control flow, he will not know where these variables are located or where he should make the CPU jump to.

Instruction Set Randomization (ISR) is also popular within this domain [6, 19]. In ISR, each system or process has its own set of instructions so that attacker-injected code will not be meaningful for the CPU of the attacked process. PointGuard [11] uses a similar principle to encrypt and decrypt all pointers within a program. If an attacker overwrites a critical pointer with his own data, the pointer, upon decryption, will be mangled, thus crashing the process instead of running the attacker's code.

Other instantiations of the randomization principle include the randomization of all data in a program's address space [8] and the randomization of the operating system's interface [10].

3.3 Conclusion

Hardening unsafe languages against the interactive attacker is a mature research area, but it is still active. The fact that the security community came up with ways to protect against the interactive attacker, fueled the evolution of attacking techniques which circumvented the proposed countermeasures. Several attacks were devised which circumvent ASLR [25, 27, 30]; return-to-libc [33] and return-oriented programming [26] defeat the non-executable stack or heap and indirect pointer overwrites can in some cases void the protection of canary-based systems [9]. Thus today, even though many of the originally-used attacking techniques no longer work, there are still scenarios which allow an interactive attacker to bypass modern countermeasures and compromise a vulnerable program.

4 Countermeasures against the in-process attacker

An attacker that can load and execute arbitrary machine code is very powerful, which makes protecting against this kind of attacker very challenging. Unfortunately, in-process attacks are also realistic, and so an important direction for low-level software security research is to protect against such attacks.

A first important class of approaches for defending against in-process attacks provides support for a trusted program to load untrusted machine code modules in its address space. A critical assumption for these approaches is that the trusted program can inspect or even modify the module before it enters the process. By combinations of code analysis and code rewriting, the newly loaded module can be sandboxed using techniques such as Software Fault Isolation [32]. These approaches are fundamentally *asymmetric*: they protect a trusted host program from untrusted modules, but modules are not protected in any way against the host. While such sandboxing is an effective technique to protect against dynamically loaded code that is potentially malicious, it can not guarantee for instance the secrecy of sensitive information in a module. Cryptographic keys used by a plug-in, for example, can still be accessed by the main application.

In order to restore the principle of source-based reasoning for both the host program as well as the module, more symmetric solutions are needed. In the last few years, two interesting lines of research results indicate potential directions. In the systems security research community, new security architectures have been developed that support the isolated execution of modules, protecting the module against its host. Research on security foundations on the other hand has shown how some of the existing low-level software security countermeasures can provide sufficiently strong protection to restore the principle of source-based reasoning, at least for simple source programming languages. We briefly discuss both research tracks.

4.1 Isolated execution of security-critical modules

Various security architectures have been developed the past few years that provide a more fine-grained protection than at the process level. The general idea is that security sensitive code and data from applications are identified and placed into different modules[1]. Each module has total control over the sensitive information it protects and specifies when and how information leaves the module. A cryptographic module for instance can prevent a private key from ever leaving the module unencrypted. Moreover, these architectures achieve this isolation with a very small trusted computing base (TCB). In particular, the TCB does *not* include the operating system.

We discuss three influential examples.

[1] The literature does not use a consistent name for this isolated code and data. Depending on the proposed security architecture, they are called AppCores [28], Piece of Application Logic (PALs) [20, 21], workloads [4], Self-Protecting Modules (SPMs) [29] etc. We will use the name *module* to refer to the general concept.

Flicker McCune et.al. [21] proposed a security architecture based on the late launch and TPM functionality present on modern computer platforms. Using a late launch sequence, the CPU can be set in a known safe state, excluding the BIOS and operating system from the TCB. After a late launch, Flicker will initialize the system and execute the module. After termination of the module, the memory allocated for the module is cleared (with the exception of the return value) and the execution of the application is resumed.

The TPM chip is used to save sensitive information between the invocation of two (possibly different) modules. The TPM chip provides secure storage based on PCR registers. These registers contain a measurement of software that was loaded. Whenever software is loaded on the system, its cryptographic hash is calculated, appended with the contents of the PCR register and the register is overwritten. The contents of these registers can never be set to a specific value. However, they can be reset either through a reboot of the entire system or a late launch sequence, depending on the type of the PCR register. When sensitive data is stored on the TPM chip, the required content of the PCR registers on retrieval of the data can be specified. This allows modules to store sensitive data for themselves or for other modules of which the measurement is known.

Flicker relies on a TCB of only 250 lines of code. Relying heavily on the slow TPM chip unfortunately also results in a significant performance overhead.

TrustVisor In subsequent work, McCune et.al. [20] reduced this overhead by several orders of magnitude. Using virtual machine extensions of recent processors, a small hypervisor guarantees the total isolated execution of modules. When a module is started, Flicker's late launch sequence is replaced with a hypervisor call ensuring that only the module is executed and cannot be interrupted. The hypervisor offers a software-based TPM implementation. It stores and retrieves a single cryptographic key from the TPM chip when the security architecture is loaded. This key can be used to store sensitive data on behalf of modules encrypted and signed on disk. As the hypervisor remains loaded in memory and is isolated from the rest of the system, the TPM chip has to be accessed only when the security architecture is loaded, resulting in a significant performance improvement over Flicker.

SICE Azab et.al. proposed [4] yet another technique to provide complete isolation of modules (called workloads) based on system management mode (SMM). SMM typically is used for system management such as power management, system hardware control or proprietary OEM-designed code. Although it is not intended to be used for general-purpose system software, its easily isolated processor environment makes it an interesting choice for a security architecture.

When the execution of a workload is requested, a system management interrupt (SMI) is issued. This causes the processor to enter a known safe state and SICE is executed. Then an isolated execution environment is prepared and the workload is executed. The authors showed that SICE has a TCB of similar size as Flicker but without the significant overhead incurred by a late launch.

Conclusions These security architectures show that it is feasible to build applications from components (modules) that live (at least conceptually) in the same process but are isolated at the machine code level, and this without relying on a trusted hosting application or operating system. The TCB is reduced to something between a few hundreds to a few thousands lines of code. Such low-level isolation mechanisms are an essential ingredient to provide strong protection against the in-process attacker, and hence are an important enabler for restoring the principle of source-based reasoning.

4.2 Fully abstract compilation

Low-level isolation mechanisms, such as Flicker, TrustVisor or SICE, are by themselves insufficient for making source-based reasoning sound. What is needed is a correct mapping of source-level protection mechanisms to low-level protection mechanisms.

Modern high-level programming languages such as Java, C#, ML or Haskell offer protection facilities such as abstract data types, the private field modifier, or module systems. These programming language concepts were designed to enforce software engineering principles such as information hiding and encapsulation. But these can also be used as building blocks to ensure security properties of programs. For instance, declaring a class instance variable private in Java protects the integrity and confidentiality of that field towards instances of other classes.

Unfortunately, these protection features are typically lost when the program is compiled. Suppose for instance that we compile a Java program to native machine code, then an in-process attacker can read or write any private variable, thus violating that variable's confidentiality and integrity. In other words, the principle of source-based reasoning fails.

However, recent research has shown that it is possible to maintain the security properties of a high-level program even after it is compiled into a lower-level language (such as native code). The way to formalize this notion of security is through *full abstraction*. Roughly speaking, compilation from a source language to a target language is fully abstract if the equivalence of source programs implies the equivalence of target programs and vice versa. That is, for full abstraction to hold, two source-level programs must be contextually equivalent if and only if their corresponding low-level translations are contextually equivalent as well. Two programs P_1 and P_2 are contextually equivalent if no third program P_T interacting with them can distinguish P_1 from P_2. At the high level, two programs can typically only interact through method calls and returns, while at the low level two programs can interact in less controlled ways, such as directly reading from or writing to each others memory locations.

The contextual equivalence of two programs can express important security properties. For instance, saying that the two classes shown in Fig.1 are contextually equivalent, is the same as saying that the value of a private instance variable in a Java class is confidential. This is obvious at the source-code level, but if we were to compile these two classes into native code using a standard

```
public class C {
  private int f = 0;

  public C() {
    [...]
  }
}
```

```
public class C {
  private int f = 1;

  public C() {
    [...]
  }
}
```

Fig. 1: Example of two contextually equivalent Java classes

compiler, their contextual equivalence would be lost. That is, the two resulting native code modules could be differentiated by a low-level test module M_T that runs in the same address space: M_T could simply inspect the memory location storing the value of `f`. The essence of a fully abstract compiler is that contextual equivalence is preserved at the low level. A fully abstract compilation scheme effectively reduces the power of an in-process attacker to that of a source code-level attacker. That is, any attack at the low level is also possible at the source code level. One can think of full abstraction as the formal equivalent in this setting to the principle of source-based reasoning.

Currently no production-class compiler for any programming language to machine code on any platform is even close to be fully abstract. However, recently two promising approaches have been proposed towards achieving full abstraction.

Techniques based on randomization Abadi and Plotkin have shown full abstraction results for ASLR [1]. At the high level they consider a simple lambda-calculus language that uses an abstract *location* type for memory locations. Each location stores a single integer that can be read or written. Some locations are public while others are designated as private, with the intent that an attacker should not have direct access to the latter. A high-level program can preserve the confidentiality and integrity of a variable by simply not exposing that variable's location.

The low-level target language is similar to the high-level language but uses integers to address memory locations instead of the abstract location type. This enables attackers to probe arbitrary memory locations. The low-level language can be considered an abstract model of a real-world Von Neumann computer architecture, as each memory location can be addressed using an integer.

To translate from the high- to the low-level language, each abstract location must be mapped to a concrete integer address. ASLR is incorporated into the low-level model by mapping private locations to *random* low-level addresses. In their paper, Abadi and Plotkin show that the security properties provided by the high-level language continue to hold at the low level, albeit in a probabilistic sense. They prove this full abstraction result for two low-level memory models. In the first model, accesses to unused addresses in memory are fatal violations that terminate the program, while in the second model such accesses are not

fatal. For the non-fatal memory access model, these results assume a bound on the number of erroneous accesses, for otherwise an attacker could iterate over all addresses.

While the lambda-calculus language used by Adadi and Plotkin is relatively simple, Jagadeesan et al. [16] have shown that the same results hold for a more complex language supporting dynamic memory allocation, first-class and higher-order references (references that can be compared and can hold other references) and control operators (the ability to perform callbacks to attacker-controlled code). These features increase the power of the attacker, as he now has an influence on the control flow of a program and can build up knowledge on the layout of memory by comparing public references. Furthermore, the extended language can model a number of system hardening principles such as instruction set randomization, enabling one to analyze their security properties in the presence of ASLR.

Even though these results are based on low-level languages that are only very rough models of a real-world low-level execution platform, they indicate that ASLR has the potential to be a very valuable technique for making the principle of source-based reasoning sound.

Techniques based on low-level memory access control Agten et al. have recently shown that it is possible to rely on low-level memory access control techniques instead of randomization, to achieve full abstraction [2]. The high-level source language for which this has been shown is a small, single-threaded object-based language with a syntax similar to Java. It supports the basic constructs expected of a modern programming language, such as branches, loops, local variables and indirect method calls (by using typed function pointers). In this language, a program consists of a number of interacting objects, each of which consists of private fields and public methods.

The low-level target language is an assembly language for an x86-like computer architecture, consisting of a program counter, a register file (including a stack pointer register), a flags register and a memory space. This basic machine model does not suffice as the target language of a fully abstract compiler. In order to support full abstraction, a program counter-dependent memory access protection scheme is added as part of the target language. This protection scheme divides memory into *protected* and *unprotected* memory, the former of which is further divided into a *code* and a *data* section. Within the code section, a variable number of memory addresses are designated as *entry points*, which are the only points through which execution of code in protected memory can start. Table 1 shows the memory access control rules enforced by this protection scheme.

Like in any object-based language, in the high-level language, the internal representation of an object is hidden from outside of that object's definition. This means two high-level objects can be equivalent from an external point of view, even though they have a different internal implementation. This maps naturally to contextual equivalence.

from \ to	*Protected*			*Unprotected*
	Entry point	*Code*	*Data*	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

Table 1: Memory permissions enforced by the low-level language

In order to achieve full abstraction, we need to use a compiler that takes advantage of the memory protection features provided by the low-level machine model. First of all, the data and code of the compiled object must be placed into the data and code parts of protected memory respectively. Next, an entry point must be created for the first low-level instruction of each method. Because the protected memory can only be entered through an entry point, one additional *return entry point* must be created to support returning from a callback (i.e. a call from protected to unprotected memory). The address of this entry point should be used as the return address for all callbacks. To prevent private data leakage or control flow tampering through the stack, the protected module must use its own *secure stack* in protected data memory. Consequently, the runtime stack must be switched from the unprotected stack to the secure stack and vice versa on each entry to or exit from the protected module. Relevant parameters and control flow information must be moved between these stacks on entry and exit points. Private data can also leak through other channels, such as the register file or the flags register. The compiler must ensure that these registers are cleared when exiting the protected module. To further preserve control flow integrity, the compiler must verify the destination of any jump to an externally supplied address (such as a callback). A valid address is either the address of the first instruction of one the protected module's own methods or an address in unprotected memory. Finally, the compiler must also ensure that any parameter value or return value passed to the protected module has a corresponding high-level value. For instance, if a boolean `false` is mapped to low-level value 0 and `true` is mapped to 1, then the compiler must verify that boolean-typed parameters and return values are confined to these values at run time.

This compilation scheme, in combination with the program counter-dependent memory access control scheme of the low-level language, has been proved to be fully abstract. Hence, with this compilation scheme, the principle of source-based reasoning holds, even for the in-process attacker model.

Implementations For this full abstraction result to have practical relevance, the program counter-dependent memory protection scheme must have an efficient real-world implementation. Both hardware and software implementations are possible. Strackx et al. [29] propose a hardware implementation for *self-protecting modules*, which uses a low-level memory protection scheme similar to the one needed to achieve full abstraction. El Defrawy et al. [13] have developed a hardware-based program counter-dependent memory protection scheme for their implementation of SMART, which is an architecture for establishing a

dynamic root of trust in embedded devices. This protection scheme provides the necessary primitives to support full abstraction and can be implemented relatively easily on current low-end microcontrollers.

For software based implementations, an interesting avenue for future work is to investigate whether the security architectures discussed in Section 4.1 can be used as building blocks. It seems likely that the kind of isolation provided by these architectures can be used to provide a suitable low-level protection mechanism as required by the secure compiler proposed by Agten et al. [2].

5 Conclusions

The field of low-level software security is an exciting and high-impact area of research. For the interactive attacker model, several decades of research have resulted in a good understanding of the problems and solutions, and some of these solutions already have found their way into mainstream operating systems and compilers.

But some important and realistic attacks against software systems are not covered by the interactive attacker model, in particular those attacks where an attacker has the possibility to load arbitrary machine code in the same process as the software under attack. An important challenge for research in low-level software security is to address this new class of attacks modeled in the *in-process attacker model*. We have shown in this paper that several important first steps in defending against this style of attacker have been taken recently.

Acknowledgments This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IBBT, IWT, the Research Fund K.U.Leuven and the EU-funded FP7-project NESSoS. Pieter Agten holds a Ph. D. fellowship of the Research Foundation - Flanders (FWO).

References

1. Abadi, M., Plotkin, G.D.: On protection by layout randomization. In: CSF. pp. 337–351. IEEE Computer Society (2010)
2. Agten, P., Strackx, R., Jacobs, B., Piessens, F.: Secure compilation to modern processors. In: CSF (2012)
3. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the 18th USENIX Security Symposium. Montreal, QC (Aug 2009)
4. Azab, A., Ning, P., Zhang, X.: Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 375–388. ACM (2011), `http://www4.ncsu.edu/~amazab/SICE-CCS11.pdf`
5. Baltopoulos, I.G., Gordon, A.D.: Secure compilation of a multi-tier web language. In: TLDI. pp. 27–38 (2009)

6. Barrantes, E.G., Ackley, D.H., Forrest, S., Palmer, T.S., Stefanović, D., Zovi, D.D.: Randomized instruction set emulation to disrupt binary code injection attacks. In: Proceedings of the 10th ACM Conference on Computer and Communications Security. pp. 281–289. Washington, D.C. (Oct 2003)
7. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX Security Symposium. pp. 105–120. Washington, D.C. (Aug 2003)
8. Bhatkar, S., Sekar, R.: Data space randomization. In: Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment. Lecture Notes in Computer Science, vol. 5137. Paris, France (Jul 2008)
9. Bulba, Kil3r: Bypassing Stackguard and Stackshield. Phrack 56 (2000)
10. Chew, M., Song, D.: Mitigating buffer overflows by operating system randomization. Tech. Rep. CMU-CS-02-197, Carnegie Mellon University (Dec 2002)
11. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the 12th USENIX Security Symposium. pp. 91–104. Washington, D.C. (Aug 2003)
12. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium (1998)
13. El Defrawy, K., Francillon, A., Perito, D., Tsudik, G.: Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. In: Proceedings of the Network & Distributed System Security Symposium (NDSS), San Diego, CA (2012), http://francillon.net/~aurel/papers/2012_SMART.pdf
14. Erlingsson, U., Younan, Y., Piessens, F.: Low-level software security by example. In: Handbook of Information and Communication Security. Springer (2010)
15. IBM: Gcc extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/
16. Jagadeesan, R., Pitcher, C., Rathke, J., Riely, J.: Local memory via layout randomization. In: CSF. pp. 161–174. IEEE Computer Society (2011)
17. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: A safe dialect of c. In: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference. pp. 275–288. ATEC '02, USENIX Association, Berkeley, CA, USA (2002), http://dl.acm.org/citation.cfm?id=647057.713871
18. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proceedings of the 3rd International Workshop on Automatic Debugging. pp. 13–26. Linköping, Sweden (1997)
19. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: Proceedings of the 10th ACM Conference on Computer and Communications Security. pp. 272–280. Washington, D.C. (Oct 2003)
20. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: Efficient TCB reduction and attestation. In: Proceedings of the IEEE Symposium on Security and Privacy (May 2010), http://www.ece.cmu.edu/~jmmccune/papers/MLQZDGP2010.pdf
21. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. In: Proceedings of the ACM European Conference in Computer Systems (EuroSys). pp. 315–328. ACM (Apr 2008), http://www.ece.cmu.edu/~jmmccune/papers/mccune_parno_perrig_reiter_isozaki_eurosys08.pdf

22. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: Ccured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst. 27(3), 477–526 (May 2005), http://doi.acm.org/10.1145/1065887.1065892
23. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
24. Robertson, W., Kruegel, C., Mutz, D., Valeur, F.: Run-time detection of heap-based overflows. In: In Proceedings of the 17th Large Installation Systems Administrators Conference. pp. 51–60. USENIX Association (2003)
25. Roglia, G.F., Martignoni, L., Paleari, R., Bruschi, D.: Surgically returning to randomized lib(c). In: 25th Annual Computer Security Applications Conference (2009)
26. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 552–561. Washington, D.C., (October 2007)
27. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the Effectiveness of Address-Space Randomization. In: Proceedings of the 11th ACM conference on Computer and communications security. pp. 298–307. Washington, D.C. (October 2004)
28. Singaravelu, L., Pu, C., Härtig, H., Helmuth, C.: Reducing tcb complexity for security-sensitive applications: three case studies. In: EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. pp. 161–174. ACM, New York, NY, USA (2006), http://www.cs.kuleuven.ac.be/conference/EuroSys2006/papers/p161-singaravelu.pdf
29. Strackx, R., Piessens, F., Preneel, B.: Efficient isolation of trusted subsystems in embedded systems. In: Jajodia, S., Zhou, J. (eds.) SecureComm. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 50, pp. 344–361. Springer (2010)
30. Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: EUROSEC. pp. 1–8 (2009)
31. Van Acker, S., Nikiforakis, N., Philippaerts, P., Younan, Y., Piessens, F.: Valueguard: Protection of native applications against data-only buffer overflows. In: Proceedings of the Sixth International Conference on Information Systems Security (ICISS) (2010)
32. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. SIGOPS Oper. Syst. Rev. 27(5), 203–216 (Dec 1993), http://doi.acm.org/10.1145/173668.168635
33. Wojtczuk, R.: Defeating solar designer non-executable stack patch. Posted on the Bugtraq mailinglist http://www.securityfocus.com/archive/1/8470 (Jan 1998)
34. Younan, Y., Joosen, W., Piessens, F.: Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Tech. Rep. CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven (2004)
35. Younan, Y., Joosen, W., Piessens, F.: Runtime countermeasures for code injection attacks against C and C++ programs. ACM Computing Surveys to appear (2012)
36. Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., Joosen, W.: Paricheck: an efficient pointer arithmetic checker for c programs. In: ASIACCS. pp. 145–156. ACM (2010), http://dblp.uni-trier.de/db/conf/ccs/asiaccs2010.html#YounanPCSPJ10
37. Zeng, Q., Wu, D., Liu, P.: Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures. In: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. pp. 367–377. PLDI '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1993498.1993541