A Compiler-Based Infrastructure for Software-Protection

Clifford Liem
Cloakware Corporation
Clifford.Liem@cloakware.com

Yuan Xiang Gu
Cloakware Corporation.
Yuan.Gu@cloakware.com

Harold Johnson
Cloakware Corporation.
Harold.Johnson@cloakware.com

Abstract

Not long after the introduction of stored-program computing machines, the first high-level language compilers appeared. The need for automatically and efficiently mapping abstract concepts from high-level languages onto low-level assembly languages has been recognized ever since. A compiler has a unique ability to gather and analyze large amounts of data in a manner that would be an unwieldy manual endeavor. It is this property that makes known compiler techniques and technology ideally suited for the purposes of software protection against reverse engineering and tampering attacks. In this paper, we present a code transformation infrastructure combined with build-time security techniques that are used to integrate protection into otherwise vulnerable machine programs. We show the applicability of known compiler techniques such as alias-analysis, whole program analysis, data-flow analysis, and control-flow analysis and how these capabilities provide the basis for program transformations that provide comprehensive software protection. These methods are incorporated in an extensible framework allowing efficient development of new code transformations, as part of a larger suite of security tools for the creation of robust applications. We describe a number of successful applications of these tools.

Categories and Subject Descriptors D.2.0 [**General**]: Protection mechanisms.

General Terms Design, Performance, Security.

Keywords *Software Protection, Tamper Resistance, Compiler-based Techniques, Code Transformation.*

1. Introduction

In today's digital world, the need for software protection is reaching its height. In application areas such as video or music, Digital Rights Management (DRM) systems are being deployed to protect the high-value content. The discovery of root keys in the system can cause a breach of valuable content; or worse, cause the release of automated exploits to release any content. In Conditional Access Systems (CAS), satellite or cable service providers are constantly bombarded by new attacks, which proliferate in our connected world and destroy a company's valuable subscription stream. In enterprise systems, the network is guarded by sets of password systems, often through hard-coded scripts. Client and server software modules become at risk to both outsider and insider attacks.

Data in-transit is only part of the security problem. The more prevalent situation is when software executes in a hostile environment, where attackers have large amounts of time and resources to spend. Hardware protection can play a part, but usually software is still needed for flexibility, upgrades, and cost effective revisions. In often circumstances, software is important for key management, revision and revocation control. How then can a distributor of software be sure that the software is robust and resistant to attack? Frequently, the platform and software are well known to an attacker with time, resources, tools and all the experts on the web. This attack landscape is often termed a *white-box* environment, where all the content is in plain sight – the opposite of a black-box environment.

A strong defense for white-box attacks can be achieved through effective build tools. A compiler has at its disposal a vast amount of semantic information about the application. By gearing transformations to specific security objectives, a compiler-based transformation tool can conceal a program's intent from both static and dynamic attacks. This tool can perform transformations that make common tampering techniques ineffective. Furthermore, when these compiler techniques are combined and interlocked with encryption and binary protection techniques, the defenses become layered and in-depth. Interlocking is a term used to describe mutual dependencies of techniques, making the counter-measure of any one technique ineffective.

This paper outlines an extensible compiler infrastructure and a number of compiler transformation techniques, the basis of a larger defense-in-depth approach. These techniques make use of known approaches in classic compilers with different goals in mind. Rather than the usual objectives of optimizing performance and program size, these techniques address a third objective axis: security. All the techniques described hereafter have been implemented and are currently in production in the compiler-based transfor-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLAS'08 June 8, 2008, Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-59593-936-4/08/06…$5.00.

mation tool known as the Cloakware™ *Transcoder*. Finally, we describe the application of the Transcoder on a real-life DRM (Digital Rights Management) application.

2. Related Work

A number of papers have been introduced to study the theory of obfuscation [2,3,26]. Although there is work that states that complete obfuscation of an arbitrary program cannot exist [3], there is also work to indicate that positive results can be achieved with obfuscation techniques [3]. This paper does not seek to prove or disprove the possibility, but rather to demonstrate a number of practical techniques that increase the difficulty of attacks, making reverse-engineering and tampering a daunting and discouraging task to any intruder.

Commercial and publicly-available obfuscators [15,16] have existed for some time; however, they typically perform simple operations such as identifier renaming, removing whitespace, reformatting, and removing unused entities. These measures usually have little effect on the final executable and merely pose a difficulty in readability.

Early work by Colberg and Thomborson [5] suggest elementary methods for obfuscation and tamper-proofing. In this paper, we describe advanced transformations built in an extensible framework aimed to resist static, dynamic and tampering attacks of the final executable. Additionally, we describe build-time diversity measures which address the space and life-cycle of security measures which provide rapid response to attacks. This includes defenses against differential attacks, the means for renewability and revocation, and the ability for meaningful revision updates.

Previous work suggests controlling the information flow through a program using the type system [25,13,27]. Labels and sub-typing are one way to denote security classes. A secure information flow can be tested through static type-checking. Other work also suggests using type qualifier language extensions [24] for sub-typing purposes. We present a sub-typed, extended qualifier system for C++ and C used for the purposes of transformed data. This is described in section 5.1.3.

Some of the techniques that we describe in this paper build upon previous work popularized by Wang et al. [7, 22], and anticipated in more elaborate form by Chow et al. [29]. They describe a control-flow transformation which arrives at a flattened control-flow graph (CFG). This type of transformation essentially produces a many-to-many basic block graph which is not easily reversed into a procedural flow-graph. In the first instance of control-flow flattening, simple integers are used to encode a switch value indicating the next block in the flow-of-control. Then, in the second instance, they improve upon this approach by placing the switch values in an array and addressing the array through pointers, essentially turning the problem into a data aliasing problem. While effective, there is still the possibility of attacks upon the array.

We also use this first instance of control-flow flattening, but then improve upon it in a much different second technique. We use encoding and decoding functions to calculate the switch value in the history of the flow-of-control. The technique is described in section 5.2.1.

3. Defining Security Objectives

Running software in a hostile environment runs the risk of having a gamut of professional and public tools being used to circumvent protections. Furthermore, any number of automated scripts and tools can be used to continually re-run software and brute force any security.

An effective approach to providing the needs of security is to support layers of protection. It considers many different facets and angles from which an attack might originate. For example, it should protect a program from static attacks. Sensitive information such as key data and strings need to be hidden from direct observation. Furthermore, any important algorithms should be concealed and obscured from a view to reverse-engineering.

Programs should also be protected from dynamic attacks. Debuggers and disassemblers are common tools used to attack a running program. Setting break-points and stepping through code is a common approach. In addition to protecting a program and data from a static attack, it is equally important to conceal and obscure the dynamic behavior of a program and its data in memory during run-time.

Among other well-known intrusions, jamming and by-passing branches are common techniques used to dynamically attack programs. Any means that protect deployed programs from modification and tampering will inhibit an aggressor. Tamper-resistance techniques are those that cause a program to behave chaotically in the presence of modification. Furthermore, the reaction and mitigation to modification can either have a hard (e.g. cause a crash) or a soft (i.e. subtle) effect. Often the latter can be more effective than the former, since it reduces the access to immediate information.

Quite commonly, programs are mass-produced as thousands and millions of copies of exactly the same binary. The *break-once, break-everywhere* exploit is a very real problem in today's connected world. An important security objective is to mitigate the wide-spread impact of a single attack. The ability to automatically create many diverse instances of the same program is a considerable advantage.

Furthermore, a related objective is the ability to react to an attack with a software revision, realizing that a differential attack is quite likely the follow-on event to a successful exploit. Making a revised program very different from an initial version, so-called software renewability is an important strategic defense.

4. A Compiler-based Transformation Framework

Any compelling implementation choices for constructing a framework for code transformations should take into account various practical requirements. For example, applicability to a wide set of processor targets implies that the retargeting process not require years of development. It is well known that good industrial compilers for languages like C++ and C take many man-months of initial development. Moreover, reliability, robustness, effective optimizations, and the ability to target a wide set of processors for commercial use is an evolutionary process that can take many years of revisions and releases (e.g. GNU GCC). Assembly code generation itself can be a sizable task given the large variation in instructions sets and ABIs (Application Binary Interfaces) for modern processors.

In an effort to maintain maximum portability, our compilation infrastructure accepts C++/C on input, but also generates C++/C on output. This property brings with it enormous advantages allowing the retargeting process to

bypass many issues regarding assembly syntax, assembly directives, ABIs, etc. On the other hand, target-specific language extensions, intrinsics, options, and pragmas must still be handled on a case-by-case basis.

With respect to transforming code for protection purposes, the generation of C++/C source code on output (as opposed to assembly code) results in very few limitations. In general, security-oriented transformations can be implemented and sufficiently described in generated code in a high-level form, low-level form, or a combination of both. The C++ and C languages are rich and descriptive languages.

There are a few limitations, however. For example, lowering C++ code to its equivalent C code implies adhering to the name mangling of the target compiler ABI to remain link-compatible with the target compiler. Link compatibility is an important advantage, meaning that protections can equally be applied to part of a system rather than necessarily the whole. For example, constraints may mean that the whole application system is not necessarily available, especially among companies where libraries are produced by distinct groups. For this reason, our compiler infrastructure can equally produce C++ on generation. However, this poses other challenges. For example, there are a number of strict ordering rules in C++ which indicate the evaluation of operations [23]. This has implication on the intermediate representations that are used.

Figure 1 shows a high-level flow diagram of our compiler transformation framework. Individual parts of the flow are described in subsequent sections.

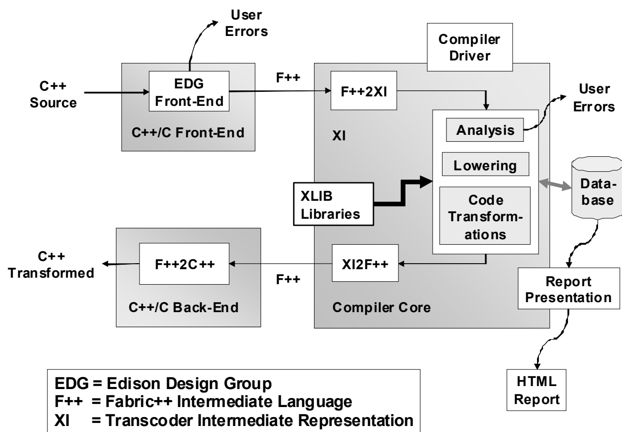

Figure 1 Compiler-based Infrastructure and Flow

4.1 Generating an Analyzable Language Representation

Our compilation framework employs the C++/C front-end by Edison Design Group (EDG) [20] from which many of today's commercial compilers are derived. From EDG's internal tree structure, we produce an intermediate analyzable language representation known as Fabric++ (Front And Back Reduced Intermediate Code for C++/C). One of the main benefits of introducing Fabric++ is to isolate the front-end and back-end components from the Transcoder core, allowing language issues to be separated from transformation engineering.

Fabric++ is a high-level and strongly-typed assembly language for a generic load/store RISC architecture. It uses a virtual register set of an infinite number, a generally capable ALU for all arithmetic, logical, and bitwise operations, and a number of well-chosen addressing modes. The addressing modes permit the representation of anything found in C++ and C, including all pointer dereferences, use of references, and use of arrays and structures/classes.

Fabric++ has both high-level and low-level control-flow constructs. Modular parts of the code such as statements, namespaces, routines, classes, inheritance, and types are all retained in Fabric++.

Virtual registers may not be addressed. This makes operations using them easily analyzable. The designation of a special virtual register called an *expression register* allows the semantics of C++ expressions to be fully retained. An expression register is one that is only ever produced and consumed once. This property makes it ideal for representing C++ expressions.

Having the properties of an assembly-level language, even though much richer than a standard assembly language, makes the wealth of today's publicly-available compiler knowledge applicable to our transformation framework. Furthermore, its language properties permit the easy manipulation of code. Unlike classic compiler tasks which include translation to low-level assembly code and optimization, security transformations may require complex manipulation of code. This includes restructuring of code, modifying types, injection/removal of code segments, and full rewriting of the code with whole program analysis.

Additionally, the generic properties of the Fabric++ language make it suitable for a Fabric++ to C++/C code generation on output, retaining the valuable high portability property. Occasionally there are behavior and syntax items which are specific to a compiler dialect; however, this is the exception and not the rule.

4.2 Analysis and Transformations

The ability to transform code from an input source to a semantically equivalent output depends on several factors:

- The ability to capture the full semantics of the language in an internal representation.
- The ability to analyze both data and control-flow, in an internal representation.
- The ability to rewrite (remove, inject, replace) the code automatically based on properties of the code and the needs of particular transformations.

These factors necessitate the employment of a rich set of structures that not only retain semantic information from the source code, but allow efficient analysis and manipulation of those structures. We have designed an internal representation for Fabric++ known as XI (i.e. the Transcoder Intermediate). This internal structure is capable of embodying Fabric++; and hence, the C++ and C source languages, in full semantic detail.

The XI representation has two principal parts:

1. The type system
2. The statement and operation representation.

The type system represents all primitive types (and their associated C++ brands), composed types, template types, self-referential types, inherited types, and any others that can be described in the very rich C++ language. Further-

more, it also represents extended types for any of the supported dialects in addition to our own extended types (described in section 5.1.3). The statement and operation structure has two distinct levels of abstraction:

1. A High Level (HL) of abstraction.

2. A Low Level (LL) of abstraction.

The HL abstraction is primarily for C++. It captures the full language with all the high-level constructs at the statement level. The complete ordering and expression level of statements and operations is maintained. If nothing else is done at this level, C++ source can be regenerated which very closely resembles the original source.

The LL abstraction follows a lowering process applied to the HL abstraction. The lowering process is defined through several operations:

- Lowered Bodies: In all executable HL bodies, high-level control statements such as while/for/do-while/if-then-else are replaced by using basic-blocks with conditional and unconditional branches.

- Local Symbol Table Merging: Local variables of nested scope are combined through a renaming process to avoid name clashes.

- Nested Lowering: Some statements and declarations are considered immovable; for example, objects of complex type, try-catch statements, setjmp/longjmp statements, etc. In these cases, HL bodies may be lowered in a 'nested' fashion, ensuring the logic between HL and LL sections is consistent.

The two levels of abstraction employed by XI permit various analyses to take place at the level to which each analysis is best suited. For example, alias analysis takes place on the High Level and deals with locations in memory and any operations where references can point to the same place in memory. Domination and data-flow analysis takes place on the Low Level and deals with operations on the abstract assembly level. Different code transformations require different dependencies, and the results of the analyses feed into various transformation capabilities which are described in section 5.

4.3 Extensible Infrastructure

A compiler infrastructure geared toward security transformations is coupled with a strong desire to quickly develop new code and data transformations to resist different and new types of attacks as they are manifested. The ability to inject code snippets into the target code for source patterns that are identified is best constructed as a generic mechanism.

We have constructed a system whereby code patterns can be described in C++ / C source and retained in intermediate format (i.e. in Fabric++ representation). Also, the mapping of these target code patterns onto source code is described in XML files as tables of data. This combination of code patterns and mapping data form internally used libraries defining specific transformations. This generic mechanism has become a great boost to productivity with respect to efficient compiler transformation development. Furthermore, code patterns described in C++ / C source provides the ability for off-line validation of the code patterns.

Figure 2 shows the make-up of internal libraries used by our compiler infrastructure. We have called the combination of Fabric++ patterns and mapping information for transformations an *XLIB* library.

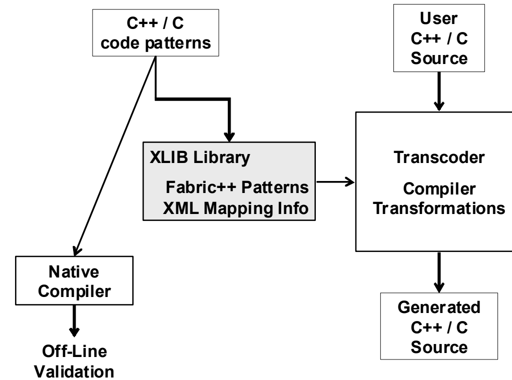

Figure 2 Internal Compiler Library Mechanism

4.4 Infrastructure Validation

The make-up of our compiler infrastructure has a great advantage with respect to validation. The infrastructure lets us validate individual modules independently. This approach is extremely valuable for the process of problem determination and reaching high levels of quality.

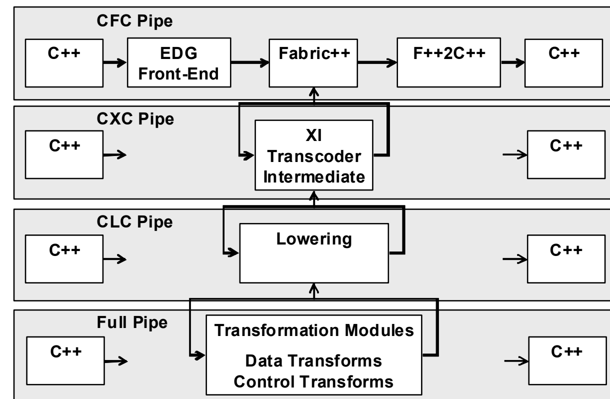

Figure 3 Compiler Infrastructure Pipes for Validation

Source code can take different paths through the compiler pipe, each pipe successively building upon the previous pipe. The pipes are defined as follows:

1. CFC Pipe
   C++ → Fabric++ → C++

2. CXC Pipe
   C++ → F++ → XI → F++ → C++

3. CLC Pipe
   C++ → F++ → XI + Lowering→ F++ → C++

4. Full Pipe
   C++ → F++ → All Transforms → F++ → C++

These compiler pipes are shown in Figure 3. Upon the original development of the Transcoder, the modules were developed in parallel; however, the validation was done in a *cascaded* manner. Only when a test had passed the CFC Pipe, would it be tested through the CXC pipe, etc. This approach accelerated the development of robust compiler modules.

Following this original development effort and when all the modules were then considered of high quality, the compiler was tested in a *reverse cascade* manner. A test is first passed through the Full Pipe. If the test fails, then it is automatically run through the CLC Pipe, etc. This approach allows us to accelerate the problem determination process.

The Transcoder is tested against the Perennial C++ and C Validation Suite [22], the Boost C++ libraries [17], the Crypto++ library [18], the GNU GCC test suite [14], the Microsoft standard, MFC, and AFC libraries, and a large number of internally developed tests. Roughly 40,000 tests currently comprise the validation suite.

5. Compiler Technology applied to Software Protection

There are two major tactics to defending software: proactive means and reactive means. Proactive tactics are concerned with the forward-facing information and generally imply using methods of concealment. Things that are considered security sensitive and need concealment in working software can include a great number of items: keys, algorithms, function call APIs, symbols, strings, data, libraries, control-flow, conditions, etc.

In contrast, reactive tactics are concerned with the behavior of software after it has been modified by an attacker. Tamper-resistance is defined as a method to alter the behavior of software depending on a detected modification.

The techniques that we employ for software protection include both proactive and reactive means, as they are complementary. We consider concealment for both static and dynamic attacks and tamper-resistance techniques for software that has been altered.

Furthermore, we present the concept of diversity as a resistance to class-attacks and resistance to differential attacks.

5.1 Compiler Techniques for Data-Flow

The heart of program functionality comes down to data in memory and calculations performed on that data. It is generally easy for an attacker to recognize known patterns of behavior, especially when it is a familiar algorithm.

We present in this section compiler transformations of the data-flow proper. This means that the familiar patterns of behavior in data-flow have been transformed into different arrangements. Furthermore, they have transformed in a diverse manner, meaning that disparate *channels* in the data-flow are transformed by different families of transformation, including different random characteristics associated with each family.

5.1.1 Alias Analysis and Data Transformations

Alias analysis [1,32] is a known technique in compiler theory, used to determine if a location in memory may be accessed by more than one instruction operation. Two pointers are said to be aliased if they point to the same location in memory. Based on alias information, we can determine all the instructions that operate on data in a common alias set.

We describe here a method to transform both data values in memory and operations from their original unencoded state to an encoded state. Locations and their corresponding operations are transformed *harmoniously*, such that the original data values need not be exposed either statically or dynamically. Examples of functions used to transform data values and operations are described in [33].

We can consider the problem of allocating data to various data encodings as an analogy to allocating data to multiple memories. A variable and set of operations can be consistently mapped to a particular data transformation in the same way it could be mapped to a specific data memory. Alias sets are imperative for the correct mapping onto data transformations. Essentially, each element of an alias set must be assigned to the same data transformation. Disjoint alias sets may be assigned data encodings independently.

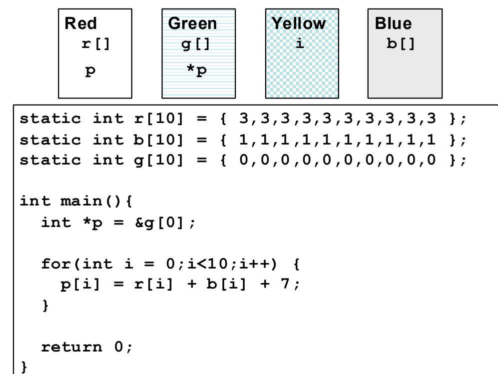

Figure 4 Data Transformation Allocation is analogous to Multiple Data Memory Allocation

The example shown in Figure 4 depicts four hypothetical data memories labeled: Red, Green, Yellow, and Blue. These are the example data transformations. When presented with a program as in the bottom of Figure 4, the compiler allocates data elements within the constraints of the program. Here, the arrays r[], b[], and g[] may be freely allocated to any memory, for example: Red, Blue, and Green, respectively. The reach of the pointer, p; however, has a constraint. It points to the array g[] and is therefore, in the same alias set of g[]. Hence, it must be allocated in the Green memory. On the other hand, the pointer, p, itself can be allocated to another memory, e.g. Red. Finally, the variable, i, can be allocated to any other memory, e.g. Yellow.

While this examples shows a finite set of data transformations (four), it is important to note that we define transformations by a formula type (i.e. family) and a set of randomly chosen constants in the formula (i.e. coefficients). This drastically increases the space to which data entities can be allocated. We have implemented a sizable number of data transform families [33], each formula containing a corresponding number of randomly chosen coefficients. Moreover, the infrastructure permits any newly conceived data transform families to be easily integrated into the Transcoder framework.

The greatest advantage to transforming the encodings of variables and operations is dynamic concealment. Not only

does the program bear a far resemblance from the original in a static sense, the program also executes much differently than the original. Successful coverage of all variable references and operations in a particular data-flow channel means that the original values of the program are never exposed in the same channel.

The impact of this result is that an aggressor who might impose a dynamic attack on a program may never see a data value that appears in the original program. From the initialization of a variable, through operations, through function interfaces and storing back into memory, the original values of the non-transformed program will never appear. This is a dynamic concealment technique that transforms all data into another mathematical space, while the program continues to maintain functional equivalence and correctness of the original program at all external points of contact. Therefore, only user-visible points of the program retain the original behavior. All internal points of the program are transformed.

5.1.2 Whole Program Analysis and Data Transformations

The effectiveness of transforming data values and operations is highly dependent on the scope to which the technique can be applied. If a program is separately compiled as a large number of disjoint libraries, then there is little opportunity for automated approaches of transforming all relevant data and operations. In a disjoint compilation setup, data values would be decoded to their original program state before crossing link boundaries to other parts of the program. This would render the dynamic concealment efforts as ineffective for the security requirement.

It is much more worthwhile to keep data encoded values in the transformed state as much as possible and over as large a scope as possible. We introduce a version of whole program analysis where information is treated globally. Through the use of a database, the Transcoder analyzes compilation units separately, and then combines the results through a global merge process. By default, the Transcoder will transform as much of the code and data space as possible through the amount of the program that is globally visible to the tool.

Adding whole program analysis to the familiar compiling stage of a build process implies a number of issues regarding the user model. There must be handling of both per-compilation unit options and global options. Especially for language like C which relies on globally linkable symbols, there must be methods of identifying large portions of code and interfaces as being non-transformed. For this purpose, header files may be identified as belonging to a *preserved domain*, with a sub-characteristic of either preserving the interface or all the code. Furthermore, incremental compilation is an important consideration. Our system handles a broad whole-program user model, enabled through the use of an extendable, generic symbolic information database.

5.1.3 Language-based Qualifier Sub-typing and Data Transformations

In addition to the automatic means that we provide for applying data transformations to a program, we also provide a number of sophisticated manual capabilities. We define a number of C++ and C language extensions for the purposes of code transformations, including both data-flow and control-flow transformations.

For data transformations, we define a number of sub-typing qualifiers [24] and casts which allow programmers to express:

- Data items that must be transformed.
- Data items that must be transformed to specific transformation families and set of characteristics.

This rich set of qualifiers allows a user to identify important assets to protect in a program, as well as apply specific transformations for specific APIs. Data that is transformed may be cast at the user's discretion and essentially be allowed to pass beyond the boundaries of the Transcoder's visibility. For example, transformed data can go into a data file, over a TCP/IP channel, be placed in the field of an XML file, or through any other data channel that traditionally passes data. By matching specific transformations on the receiving side of the communicating interface, the transformed API is made compatible. Furthermore, the mapping and diversity options of the compiler allow even these specified transformations to be altered at build time to allow a varying (though consistent) API to be created.

The C++/C language extension set for code transformations includes three data type qualifier extensions as follows:

- *_xc_transform* Specifies a data transformation to an unspecified transformation type.
- *_xc_transformtype(x)* Specifies a data transformation to a specific logical transformation type, x.
- *_xc_preserve* Specifies that the data variable is to be preserved in its original form.

These qualifiers may be placed just as C-V (const/volatile) qualifiers, modifying the type of a data declaration. Each of the data type qualifiers is mutually exclusive. At most, one qualifier may be placed on a particular entity.

Foster et al. [24] observe that type qualifiers introduce a sub-typing relationship. Given a base type, *T*, the above introduced data type qualifiers form the following sub-typing relationships:

$$T \le _xc_transform\ T \le _xc_transformtype(x)\ T$$
$$T \le _xc_preserve\ T$$

The *_xc_transform* qualifier on a type *T* forms a sub-type of the type *T*. Additionally, the *_xc_transformtype(x)* qualifier forms a further sub-type of the type *_xc_transform T*. The sub-typing relationship moves from the most general case, on the left, to more specific cases on the right.

_xc_preserve T is a special sub-type that may not coexist with *_xc_transform* or *_xc_transformtype()*. It indicates that the data type is not to be transformed at all costs.

The sub-typing relationship forms the basis for all of the possible conversions between the types and qualified types. A type *T* may be promoted to the type *_xc_transform T*, but not vice-versa. The same relationship holds for *_xc_transform T* ≤ *_xc_transformtype(x) T*. This is equally true for level skipping qualifications: e.g. *T* ≤ *_xc_transformtype(x) T*. In simple terms, this means that a base type is the most general; the type can become more specific by being a generally transformed type; finally, the type can become further specific by being a specific logical transformed type.

When a data entity is assigned to a logical transform type using *_xc_transformtype(x) T*, the identifier *x* is mapped externally to data transform characteristics. This is done through a command-line interface or a configuration file, which allows project-wide options to be specified. The data transform characteristics are family types and coefficients, described earlier.

The C++/C language extension set also includes a data cast operator: *_xc_transformcast<T>*.

- *_xc_transformcast<T> / xc_transformcast(T)*

  Perform a cast from one transformation type to another transformation type. No conversion of the type is performed (similar to the C++ reinterpret_cast<T>).

There are two forms of the data-cast that are equivalent, one with angle brackets (i.e. <>) and one with round brackets (i.e. ()). The round bracket form is for the C language. The *_xc_transformcast<T>* operator allows a transformed type to be cast directly to another transformed type (including a preserved type). This allows such features as allowing tranformed values to go beyond the transformation boundary as well as allowing testing of data transformation features.

This section, as an example, has described a subset of the C++/C language extension set that we have defined for code transformations. There are a number of function level, class level, and operation-centric extensions that are not described here.

5.1.4 String Transformations

Strings are a common point of attack for an executable program, since they are a convenient anchor point that an intruder can easily understand. Whether in ASCII or wide char form, all the string literals in an executable can easily be retrieved by programs like the UNIX program *strings*, or with a disassembler like IDA Pro [19] or OllyDbg [21].

While strings are just arrays of integer data (e.g. char, short, int), the attack model is slightly different than that of regular data. It is important to transform all the strings in a program with general coverage to obscure any initial points of attack.

For each string in a program, we define a compiler transformation where the original values are initialized in a concealed state. These are decoded at first use in the program through a variety of means. Once decoded, a string available for use in the program.

In addition to the general coverage of string transformations, data transformations may also be combined with string transforms where applicable. String transformations are not intended as a strong concealment, but simply as a broad coverage which complements other techniques that hide assets.

5.2 Compiler Techniques for Control-Flow

In general, classic compilers interpreting a high-level language turn control-flow constructs including loops, call/return statements, and conditional statements into a predetermined set of conditional and unconditional branches for a given target instruction-set. While there are a large number of optimizations related to control-flow, the mapping onto available branch instructions is generally a deliberate and arranged set of steps with disposition given only to the impact on final performance and size.

If a program originates from a well-structured high-level program, reverse engineering the control-flow of an executable is generally a simple matter of disassembling instructions and repositioning blocks of code. Decompilers and disassemblers often provide this capability directly.

In a white-box context, providing security is a matter of both protecting and concealing the intent of the control-flow. Concealment can be achieved through a number of code transformation methods. Protection can be achieved through tamper-resistance methods.

5.2.1 Control-Flow Analysis and Resistance to Reverse Engineering

Within a function of a program, the control-flow consists generally of a set of loops, conditional, and unconditional branches. A classic compiler translates the high level statements of the program into instructions available in the processor in a predictable manner. As described in [7,8], control-flow flattening is a process of transforming a well-structured control-flow into a controlling loop and switch statement. A switch variable then controls the flow of control from basic block to basic block. In other words, the original control-flow is transformed into a data directed control-flow.

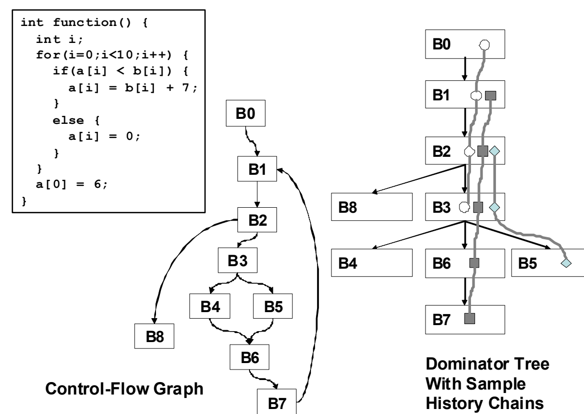

Figure 5 History Chains based on Dominance Property

We use the technique described in [7, 8] as a basis for new advanced techniques. Starting with the original control-flow graph (CFG), a dominator tree [1, 28] can be calculated. This graph indicates which basic blocks dominate other blocks and conversely which basic blocks post-dominate other blocks. This information can be used to calculate history chains, where the flow-of-control must pass in the program. History chains can be characterized by their length and domination information.

We define a set of functions, *H1 ... Hn*, which calculate the values for controlling a switch variable, k, for control-flow flattening. These functions are placed in appropriate basic blocks of the history chain. The first of these functions is an initialization, while the subsequent functions are a series of encoding and decoding functions, which eventually arrive at the target switch variable value needed to direct the flow-of-control. A sample set of steps is as follows:

1. H1: Init (k) initialization of k
2. H2: Enc^1 (k) encoding function 1 of k

3. H3: Enc^2 (k) encoding function 2 of k
4. H4: Dec^2 (k) decoding function 2 of k
5. H5: Dec^1 (k) decoding function 1 of k

The above example shows a series of five functions: an initialization and two pair of encoding/decoding functions which calculate a value for the switch value, k. An unlimited and diverse set of these functions can be created which vary according to operations, constant coefficients, and number of functions.

We match a history chain of basic blocks with a set of functions for calculating the switch value. Each basic block in the history contains a call to an encoding/decoding function. For example, a history chain of length 5 will be matched with a set of 5 functions.

Figure 5 shows a sample C function, its corresponding Control-Flow Graph (CFG), and the dominator tree that has been calculated from the CFG. Also depicted are three sample history chains through the dominator tree of ending in block B3 (history chain of length 4), block B5 (history chain of length 3), and block B7 (history chain of length 5). These history chains can be used to compute the control-flow flattening switch variables with functions, *H1… Hn*. Likewise, the process can be repeated for all basic-blocks, except for the entry block B0, which has no basic-block history that precedes it. Other means are used to conceal the computed switch value for the entry block.

Once the function is transformed into Control-Flow Flattened form, the history chains become much less obvious. It is not possible to statically retrieve the intended flow-of-control without understanding of the functions, *H1... Hn*.

Placing the calculation of switch values in the history of the CFG effectively conceals information regarding the flow-of-control itself. An aggressor looking to reverse engineer the control-flow of a program is forced to look beyond a static analysis of the program.

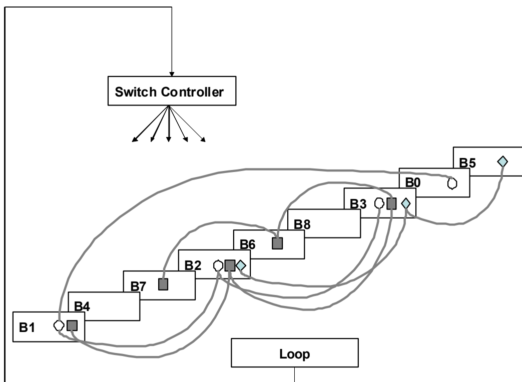

Figure 6 Control-Flow Flattening + History Functions

Figure 6 depicts the sample in flattened form. The basic blocks are no longer in an ordered layout, but effectively placed along side of each other. The history functions are interspersed in the blocks, making reverse engineering a difficult problem for the attacker.

5.2.2 Analysis of Conditionals and Resistance to Branch Jamming

A common technique of attacking code is to circumvent conditionals by *jamming* or bypassing the condition. On the assembly level, jamming a branch is the simple matter of replacing a conditional branch with a NOP or an unconditional branch. The attack then comes down to the identification of a suitable conditional branch instruction.

We introduce a tamper-resistance technique known as Branch Protection. This is a method whereby the compiler transforms the code relevant to the condition and branch, such that a jamming attack will have an adverse affect on the program. The aim of the approach is to insert dependencies throughout the program which depend on the truth of the condition. If the condition is not true, jamming the branch may cause an undesired set of blocks to execute; however, operations in those blocks also have dependencies on the condition and will not execute as the attacker has expected.

The approach begins with an analysis of the relevant parts of the program relating to the condition and branch. The following items are identified:

- Sources of the condition.
- Basic blocks that dominate of the condition.
- Basic blocks that post-dominate both target blocks of the condition.

We then draw on a set of library functions which have a number of special properties. We define a condition function, *P*, which takes the following as parameters:

1. The condition type.
2. The sources of the condition.
3. A target variable or expression from the program.

In normal operation, the function, *P*, calculates the same value for the target variable or expression given in point 3. In other words, the value is simply passed through the function. In a case where the condition does not hold, the function calculates a random large or small value that does not equal the original target value.

We have defined a diverse set of functions, *P*, that reside in a compiler internal library as described in section 4.3. The functions are used during compilation as a result of a number of automatic and manual controls. The user may specify highly sensitive branches with extended keywords, as shown in Figure 7 (i.e. _xc_protectif), as well as the number of dependent target variables or expressions, and also specific target variables or expressions. If the targets are not specified explicitly, the compiler determines candidates in post-dominating basic blocks through a heuristic. The target variables or expressions need not be directly in the blocks to which the conditions are branching, but can be further away in the dominator tree. The greater distance from the condition to the dependent targets makes it generally more difficult for an intruder to determine the relationship.

The Branch Protection feature is continually tested with a number of automated tests. In addition to the feature itself, we have devised an automatic means to simulate a branch jamming attack. Therefore, a program transformed with the Branch Protection feature can be tested with and

without a simulated branch jamming attack. The former is expected to behave the same as the unaltered program. The latter should behave adversely. The adverse behavior can be categorized as either a hard failure (e.g. a segmentation fault or core dump) or a soft failure (e.g. a value that gets altered and causes the program to behave differently in a subtle manner). Both of these behaviors are a valuable defense to software attacks.

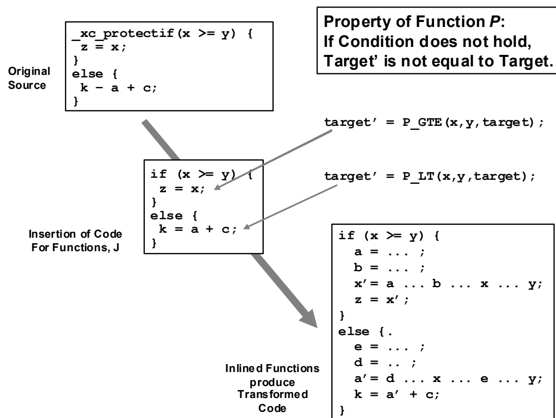

Figure 7 Applying Functions for Branch Protection

5.2.3 Inter-procedural Call-Graph Verification

In addition to single function techniques and data transform techniques across an application; we also introduce techniques to verify the call-graph of an application. This builds upon the whole program infrastructure described in section 6.

We first compute a set of may-call and must-call graphs for the full application. This information is arrived at by combining a flow-sensitive data flow analysis with the static call-graph. The information is merged in an incremental fashion, such that changes to any compilation unit will trigger only local updates in the compiler database.

Armed with this call-graph information, a number of protection behaviors are inserted which create dependencies along call-graph chains. Although we leave out the details here, the principle resembles the history functions described in section 5.2.1. If any functions are *snipped* out of the application, they cannot function in an isolated manner, since they are dependent on other functions in the call-graph.

5.3 Optimizing Transformed Code

Given that the Transcoder is a source-to-source tool and the target compiler is always subsequently invoked, it is tempting to delegate all optimizations to the native compiler. However, many of the security-oriented transformations described earlier can inhibit the effectiveness of standard optimization. For example, the control-flow flattening technique described in section 5.2.1 will naturally diminish the effectiveness of a flow-sensitive data-flow analysis needed for data optimizations. Similarly, a program in the flattened form cannot be easily analyzed for domination information. Many optimizations are based on the dominator tree.

In general, the Transcoder performs compiler optimizations after the initial analyses phases and prior to transformation like control-flow flattening. We have concentrated our efforts initially on standard optimizations which would be effective for code that resides in xlib form and is being injected into user's code. However, we expect that many other standard and aggressive optimizations would be extremely effective as well [1,6,28,32].

Since the data transformation technique (section 5.1.1) results in the injection of inlined functions with several constant coefficients, we have implemented constant propagation, constant folding, and common sub-expression elimination. Furthermore, given our unique intermediate representation to retain expressions in C++ (section 4.1), we have also implemented optimizations for inlining special property functions into expression trees. We have noticed that a larger expression tree in generated code results in faster code when compared to code that uses many temporary variables.

We envision implementing many standard and new optimizations for transformed code in the future [1,6,28,32]. We are not able to rely on target compiler optimizations given the impact of rewriting the code with security-oriented transformations.

5.4 Build-Time Diversity, an Enabler

Inevitably, even the strongest, most secure software systems get compromised. Since this is the premise, a good approach is to prepare a defense which minimizes the impact of a breach. A diverse set of programs which offer the same functionality reduces the impact of a class attack. In addition, an automated attack which creates an exploit to one instance of the program does not necessarily affect its diverse counterparts.

All of the compiler-based techniques described in this paper have an additional diversity capability built-in. When a technique is presented with an equivalent choice, it uses a function based on a Pseudo-Random Number Generated (PRNG) to make the choice randomly. Examples of equivalent choices are as follows:

- Choice of encoding / decoding function families and coefficients in Control-Flow Flattening, Inter-Procedural Call Verification, Data Transforms and String Transforms.
- Order and layout of blocks in code generation.
- Choice of constants (e.g. switch values in Control-Flow Flattening).
- Ordering of function parameters in Global Transcoding

Making random choices will result in different instances of an original program with distinctive and unique program structures. All the transformation capabilities have diversity built in; however, it is important to note that this is driven through a seeded PRNG making each instance reproducible.

Using diversity, it is possible to create a wide range of differing, yet functionally equivalent binaries for a given program. This capability can be used to create a large set of instances which is immune to an automated class attack. Furthermore, diversity can be used in many scenarios to achieve a wide range of results. A few simple examples are described hereafter.

Consider a software system with a revision of software that was compromised. A new revision of the software with new functionality can be produced, but furthermore it can

be made diverse from the revision that got compromised. Any modules that remain from the earlier version would be altered in structure. This makes a *differential attack*, one that relies on comparing the revisions of software, much less effective.

Consider a software system where a main module (e.g. executable) communicates through an API to a sub-module (e.g. DLL/shared library). Suppose we create 2 diverse instances of the sub-module for 2 different customers. With a careful construction of the API and diversity settings, it is now possible to revoke 1 customer while retaining compatibility for the second customer, simply by an update of the main module.

These are just two of the ways in which diversity can be used to achieve software security. There are many more. Software diversity is an enabler for:

- Resistance to Differential Attacks
- Renewability
- Revocation
- Response to Compromised Software.

A system designed with security in mind should consider much more than a single revision of software; it should consider the entire deployment space, including the range of customers, as well as the security life-cycle.

5.5 Build Integration

The addition of security-oriented tools in a pre-compile, post-compile, or post-link position significantly alters the steps on building applications for development, test, and distribution. These tools throw into question the traditional method of functional debugging and quality assurance.

We recommend breaking the development process into distinct phases, each with discrete goals:

1. Functional Development and Testing
2. Security Analysis and Application of Protection
3. Security / Performance Tuning
4. Penetration and Resistance Testing

As in any development process, each of these phases is cyclic in nature and project requirements can cause jumps from one phase to another. However, in general, these phases should be approached in sequence, each with a characteristic objective.

Phase 1 of this development process is concerned with the functional mechanics of the system. Debugging and testing the function of the application is the primary objective. While some aspects of security may naturally fall into functional development including key flow, the 2nd phase is the main step for security analysis including modeling threats and building attack trees. Prioritization of the highest value assets are an important part of this 2nd phase. During this step it is also important to build in the ability to parameterize security provisions according to priority. These parameters are an important part of the 3rd phase, where security/performance tuning takes place. Finally, the last phase is where actual attacks are done on the application in an attempt to circumvent the protection.

Regarding the subject of debugging, it is clearly a great difficulty to debug a program where code transformations have already been applied. This is the reason we stress the importance of validation of tools (see section 4.4). Validation not only includes testing security transformations, but also testing a large sample of diverse instances for each transformation (see section 5.4). This is also the reason we stress the separation of functional verification separately from the application of protection.

6. Application and Case Study

The compiler techniques in this paper have been implemented in a security tool known as the Transcoder. The Transcoder is part of security suite of tools which also includes tools for binary level protection and white-box cryptography capabilities.

Both the Transcoder and the full set of security tools have been applied to a wide range of applications. These include several Digital Rights Management (DRM) systems, Conditional Access (CA) systems, password management systems, game systems, video systems, home automation systems, and network management systems.

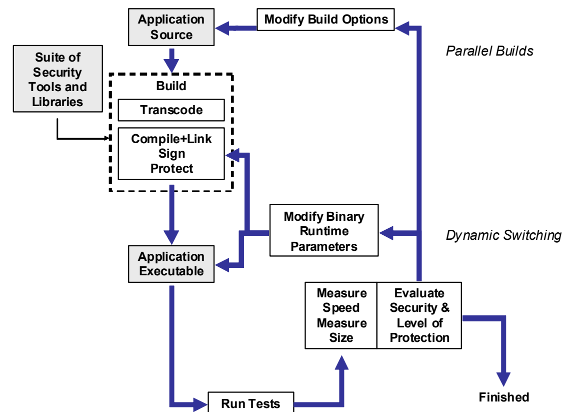

Figure 8 Security / Performance Tuning Cycle

The application of the tools to a given system begins with a security analysis step, identifying and prioritizing the assets and security sensitive areas of the system. The tools are then applied, with priority given to the security assets. Typically, the next step is to run through a performance/security tuning exercise, which trades off time and space against strength of defense to meet an optimal balance for the requirements of the application. Finally, when the product has met security robustness and performance expectations, the last step is resistance and penetration testing. This last step provides the security assurance needed to allow the system to go to production. The entire process time varies with the complexity of the system; however, a typical process can range from 2 to 10 weeks.

As an example, we outline the application of the security suite to a DRM system. This is a pre-existing digital rights management system where licenses for media are passed over a network in the form of certificates. These certificates contain keys for media content. The certificates are encrypted by a license server and passed to client software. The client software must securely decrypt the certificates and determine the system has the rights to play the media.

The details to the system will not be described here; however, the process to secure it will be outlined. The first step is a security analysis of the system. The critical assets are first identified and an attack tree [4] is created. The

branches of the attack tree are then protected using tools and techniques from the security suite. These techniques include all of the transformation means described in this paper.

Figure 8 shows the security/performance tuning cycle taken to produce the final DRM product. At the left of the diagram is the usual build process, with the addition of security tools found in the tool-suite. These include the Transcoder, which contains the source-to-source compiler transformations described in this paper, plus a number of binary level tools not described here. The bottom part of the diagram shows the running of the application tests and measurements of application run-time and application binary size alongside an empirical evaluation of the security achieved by the particular build settings. The cycle on the right side of the diagram includes modifying runtime parameters (dynamic switching) as well as duplicating the build of the application with modified settings (parallel builds). The latter need not be a sequential process; parallel builds can be launched with the availability of machines and disk space. The tuning cycle allows a large number of application builds to be evaluated for their performance versus security settings.

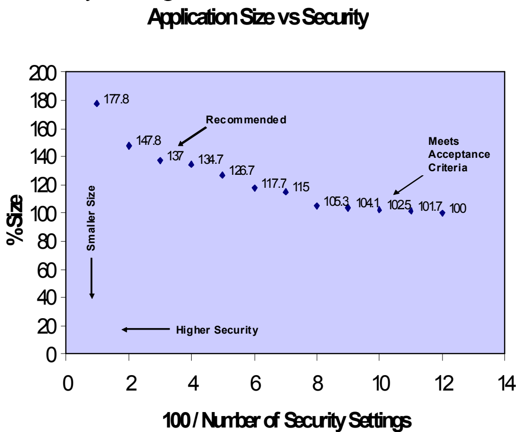

Figure 9 Application Size versus Security

Figure 9 shows the evaluation of a number of application build instances. The Y-axis shows the size of the final binary as a percentage of the original program without protection. The X-axis is a scale which is reverse-proportional to the number of security settings that have been applied. A lower number means that more security transformations have been applied to the code base.

Each point of the graph represents a different solution produced by applying a set of build-time options. The build-time options are determined through the security analysis on different assets in the system and the attack tree analysis. Higher security considerations are given to assets which can cause larger impacts. For example, in this case, a device key is considered the root of trust and highest priority, while other derived keys are considered to have less impact, since they may guard only specific content. More transformation option considerations will be placed on code with a higher impact of an attack.

Typically, a set of parallel builds with various build options will result in a *cloud* of solutions on the size versus security graph. The lower, left-hand edge of the graph, corresponding to smallest size and most security, contains the most interesting solutions. The rest of the solutions are dismissed as non-optimal. The ideal solution, which is unattainable, is at the origin of the graph.

Figure 9 shows many possible solutions of which two were considered acceptable for production. One meets an acceptance criterion of 150% of the original size of the application. This is indicated by the solution labeled 102.5. The actual in-memory of size of this solution is 1.4 x 102.5 = 143.5, accounting for the compression/decompression done at loading. The figure also shows a second solution labeled 137 (i.e. final inflated, in-memory size = 191.8), indicating recommended security settings covering more sensitive assets and algorithms with further transformations.

Figure 10 shows the evaluation of the application run-time versus security. This is a similar analysis to the application size versus security analysis. The Y-axis shows the application run-time as a percentage of the original, unprotected application run-time. The X-axis is a scale which is reverse-proportional to the number of security settings that have been applied.

In this case, the solution labeled 140 meets the acceptance criteria of less than 150% of the original application run-time. A second solution with recommended security settings results in a speed that is 240% of the original application.

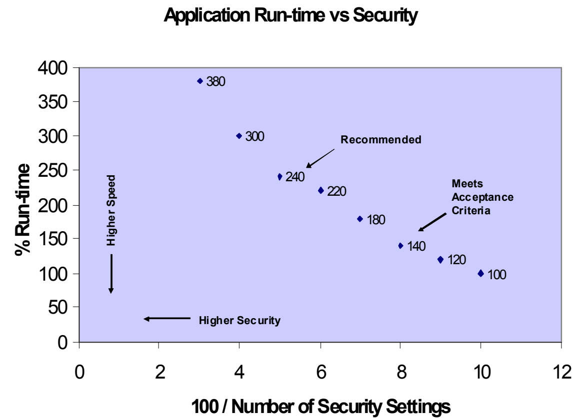

Figure 10 Application Speed versus Security

While the analysis presented here for the DRM application leaves out many of the details regarding security settings, build options, and test set-ups, etc., it does illustrate the methodology and security/performance tuning process. In the end, the lesson is that security comes at a cost to performance. In the case of this application, the recommended security settings come at a size penalty of nearly 2 times (137% in compressed form), and a speed penalty of nearly 2.5 times the original unprotected form. Given that the DRM certificate handling happens once for the playing of media content, this may not be a critical overhead.

7. Conclusion and Future Work

This paper has presented an extensible compiler-based infrastructure for code transformations for the goals of software protection. We have described our source-to-source framework, intermediate representation, internal library structure, and validation methodology. The applicability of known compiler analysis techniques have been shown in this framework.

A number of data-flow and control-flow techniques have been described for software protection in a white-box envi-

ronment. We describe a number of automatic and manual controls for these transformations in our compiler environment. We also discuss the uses diversity in the context of upgrades, revocation, and renewability.

Finally, we describe a case-study using our security suite for a DRM application. Trade-offs of security versus size/speed are discussed. The conclusion is that security does come at a price. It affects both size and performance of the final solution. On the other hand, protection of the application is a trait that should not be overlooked. Effective design engineering will trade off all three aspects wisely and objectively: size, performance, and security.

This paper has drawn a particular emphasis to existing source-level transformations. There are a number of new transformations under development. While we recognize that these techniques are an important part of software protection, there are additional security benefits to binary level techniques such as anti-debugging, integrity verification, secure loading, and code encryption. These binary techniques complement source-level transformations, working together to provide a multi-layered defense that provides much stronger protection than source-level only or binary-level only code protection. This is especially true when these methods are interlocked, meaning that an exploit of one technology is ineffective because of dependencies on another technology. Future work in this area will include orchestrating the effectiveness of both source-level and binary-level technologies, and enhancing our tools to cover all steps in the build process including compilation, building libraries, and linking. Additionally, we will continue to enhance work in the area of white-box cryptography effectively using these source-level and binary-level techniques.

Acknowledgments

The authors acknowledge contributions made by the following people: Marco Jacques, Vlad Vieru, Bahman Sistany, Robert Durand, Jonathan Emmett, Roy Germon, Lin Li, Andrew Szczeszynski, Phil Eisen, Dan Murdock, Yongxin Zhou, and Alec Main.

References

[1] A. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison Wesley, 1988, 796 pp.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang, "On the (Im)possibility of Obfuscating Programs", CRYPTO 2001, pages 1-18.

[3] B. Lynn, M. Prabhakaran, and A. Sahai. "Positive Results and Techniques for Obfuscation", Proceedings of Eurocrypt, 2004.

[4] B. Schneier, "Attack Trees, Modeling Security Threats", Dr. Dobb's Journal, Dec, 1999.

[5] C. Collberg, C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection", IEEE Transactions on Software Engineering 28:8, 735-746, Aug. 2002

[6] C. Liem, Retargetable Compilers for Embedded Core Processors: Methods and Experiences in Industrial Applications, Kluwer Academic Publishers, 1997, 155 pp.

[7] C. Wang, J. Davidson, J. Hill, and J. Knight. "Protection of software-based survivability mechanisms". In International Conference of Dependable Systems and Networks, Goteborg, Sweden, July 2001.

[8] C. Wang. "A Security Architecture for Survivability Mechanisms". PhD thesis, Dept. of Computer Science, Univ. of Virginia, Oct 2000.

[9] C. Fischer, R. LeBlanc Jr., Crafting a Compiler with C, The Benjamin/Cummings Publishing Company, Inc., 1991, 812 pp.

[10] D. Aucsmith. "Tamper resistant software: an implementation. Information Hiding", Lecture Notes in Computer Science, 1174:317{333, 1996.

[11] David W. Price, Algis Rudys, and Dan S. Wallach, "Garbage Collector Memory Accounting in Language-Based Systems", 2003 IEEE Symposium on Security and Privacy (Oakland, California), May 2003.

[12] G. Barthe, D. Naumann, and T. Rezk, "Deriving an information flow checker and certifying compiler for Java", Proceedings of Symposium of Security and Privacy'06. IEEE Press, 2006.

[13] Heiko Mantel and Alexander Reinhard, "Controlling the What and Where of Declassification in Language-Based Security", In European Symposium on Programming (ESOP).LNCS, 4421. Springer, 2007.

[14] http://gcc.gnu.org/install/test.html

[15] http://preemptive.com/

[16] http://proguard.sourceforge.net/

[17] http://www.boost.org/

[18] http://www.cryptopp.com

[19] http://www.datarescue.com/

[20] http://www.edg.com

[21] http://www.ollydbg.de/

[22] http://www.peren.com/

[23] International Standard, ISO/IEC 14882:2003(E), "Programming languages — C++"

[24] J. Foster, M. Fähndrich, and A. Aiken, "A Theory of Type Qualifiers]", ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99). Atlanta, Georgia. May 1999.

[25] L. Zheng, A. Myers, "Dynamic Security Labels and Static Information Flow Control", International Journal of Information Security, 6(2–3), March 2007. Springer.

[26] M. Dalla Preda and R. Giacobazzi, "Control Code Obfuscation by Abstract Interpretation", In Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05). pages 301-310, IEEE Computer Society Press.

[27] R. Giacobazzi and I. Mastroeni, "Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation", In 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04), pages 186-197. ACM press.

[28] Robert Morgan, Building an Optimizing Compiler, Butterworh-Heinemann, 450 pp.

[29] S. Chow, H. Johnson, Y. Gu, "Tamper Resistant Software – Control Flow Encoding", US Patent 6,779,114, filed August 19, 1999.

[30] S. Chow, P. Eisen, H. Johnson, and P. van Oorschot. "A White-Box DES Implementation for DRM Applications". In Proceedings of 2nd ACM Workshop on Digital Rights Management, Nov 18 2002.

[31] S. Chow, Y. Gu, H. Johnson, V. Zakharov. "An approach to the obfuscation of control-flow of sequential computer programs". In G. Davida and Y. Frankel, InformationSecurity, ISC 2001, vol 2200 of Lectures Notes in Computer Science, Springer.Verlag, 2001. 68.

[32] S. Muchnick, Advanced Compiler Design and Implementation, Elsevier, pp 856.

[33] Y. Zhou, A. Main, Y. Gu, H. Johnson, "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms", 8th International Workshop on Information Security Applications (WISA 2007), pp 61-75, Springer LNCS 4867, 2008